AD-A035 708 NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF F/G 17/2

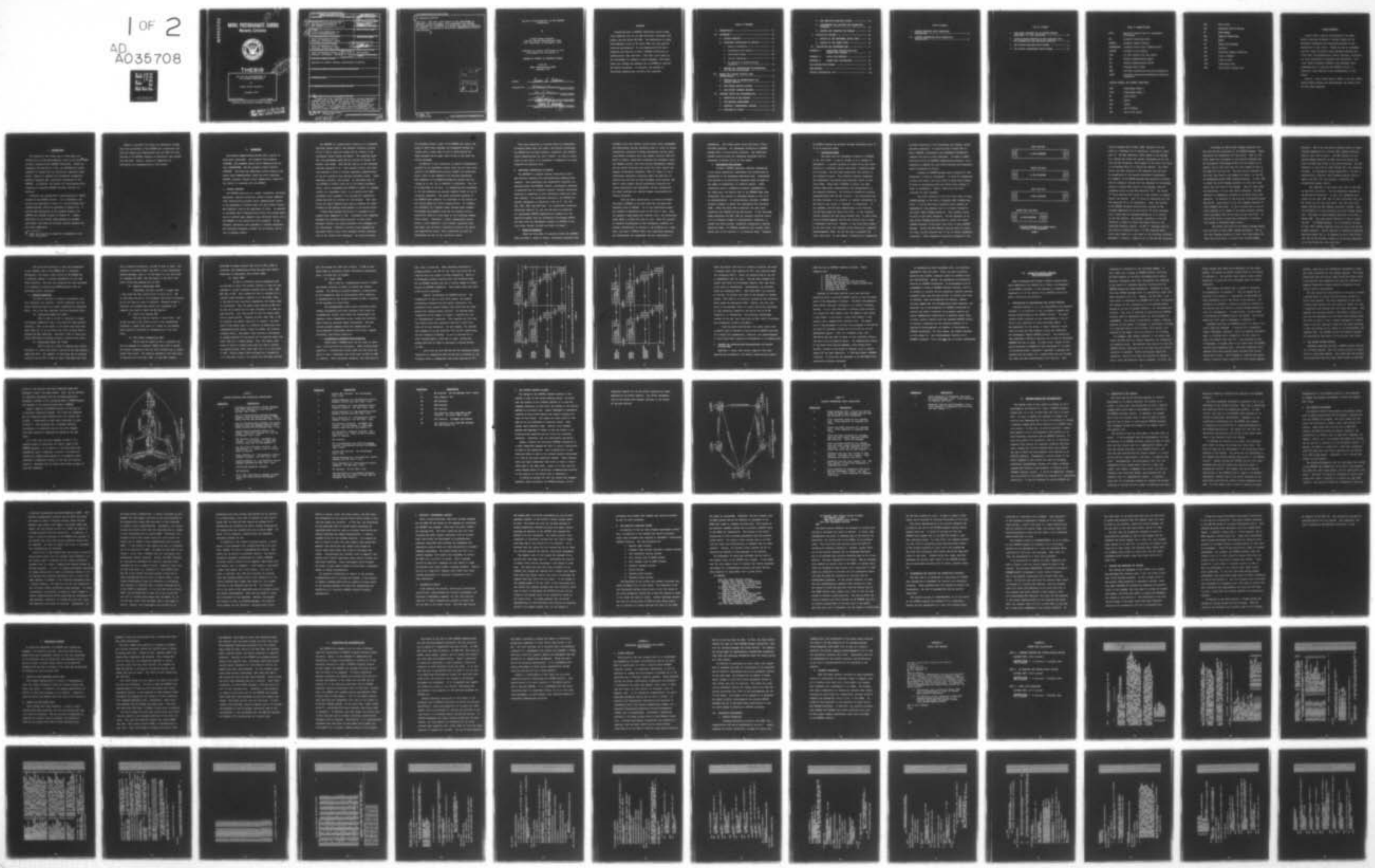
THE USE OF MICROCOMPUTERS IN DCS AUTODIN TRIBUTARIES.(U)

DEC 76 G E ANDERSON

UNCLASSIFIED NL

1 OF 2

AD A035708



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ADA035708

(2)
B.S.

NAVAL POSTGRADUATE SCHOOL

Monterey, California

D D C
FEB 17 1977
C

THESIS

THE USE OF MICROCOMPUTERS IN
DCS AUTODIN TRIBUTARIES

by

Gordon Ernest Anderson

December 1976

Thesis Advisor: V. M. Powers

Approved for public release; distribution unlimited.

COPY AVAILABLE TO DDC DOES NOT
PERMIT FULLY LEGIBLE PRODUCTION

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) 6 The Use of Microcomputers in DCS AUTODIN Tributaries,		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis, ~~December 1976~~
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Gordon Ernest Anderson		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	11	12. REPORT DATE Dec[redacted]r [redacted]76
		13. NUMBER OF PAGES 118
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Naval Postgraduate School Monterey, California 93940	12 118p.	15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Present-day Mode I AUTODIN tributaries utilize large-scale computers such as the IBM 360 series, Burroughs 3500 series, and the Univac DCT 9000. The feasibility of using micro-computers (such as the Intel 8080) for such applications was investigated. It was demonstrated that microcomputers can function as Mode I AUTODIN tributaries at speeds greater than

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

251450

20. Abstract (Cont'd)

2400 baud. This fact could result in the replacement of expensive leased equipment with subsequent cost savings and expanded use of AUTODIN in tactical and mobile situations. In addition, new methods of describing communication protocols were explored.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

The Use of Microcomputers in DCS AUTODIN Tributaries

by

Gordon Ernest Anderson
Captain, United States Marine Corps
B.S., University of Washington, 1968

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1976

Author: Gordon E. Anderson

Approved by: V. Michael Powers
Thesis Advisor

Uno R. Kodres
Second Reader

[illegible signature]
Chairman, Computer Science Department

David A. Schrady
Dean of Information and Policy Sciences

ABSTRACT

Present-day Mode I AUTODIN tributaries utilize large-scale computers such as the IBM 360 series, Burroughs 3500 series, and the Univac DCT 9000. The feasibility of using microcomputers (such as the Intel 8080) for such applications was investigated. It was demonstrated that microcomputers can function as Mode I AUTODIN tributaries at speeds greater than 2400 baud. This fact could result in the replacement of expensive leased equipment with subsequent cost savings and expanded use of AUTODIN in tactical and mobile situations. In addition, new methods of describing communication protocols were explored.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

TABLE OF ABBREVIATIONS

ASCII	American Standard Code for Information Interchange
ASC	Automatic Switching Center
AUTODIN	Automatic Digital Network
AUTOSEVOCOM	Automatic Secure Voice Communications
AUTOVON	Automatic Voice Network
baud	(in this thesis) bits per second
DCA	Defense Communications Agency
DCS	Defense Communications System
LMF	Language Media Format
UART	Universal Asynchronous Receiver/Transmitter
USART	Universal Synchronous/Asynchronous Receiver/Transmitter

AUTODIN CONTROL AND FRAMING CHARACTERS:

ACK1	Acknowledge Number 1
ACK2	Acknowledge Number 2
BP	Block Parity
CAN	Cancel
DEL	Delete
EM	End of Medium
ETB	End of Text Block

ETX	End of Text
INV	Suspected Invalid Message
MC	Mode Change
NAK	Negative Acknowledge
REP	Reply
RM	Reject Your Message
SEC	Security
SEL	Selection Channel Characters
SOH	Start of Header
STX	Start of Text
SYN	Synchronous Idle
WBT	Wait Before Transmitting

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis advisor Assistant Professor V. Michael Powers for his invaluable assistance and stimulating guidance in the preparation of this thesis. Thanks are due to Lieutenant (jg) Gail Junge and Senior Chief Radioman Frederick Guth of the Naval Telecommunication Center, Monterey, California, for their enthusiastic cooperation and assistance. Also, I must thank Lieutenant Commander Jane Renninger whose pioneering work in reducing the AUTODIN protocol to transition state machines proved indispensable to this thesis.

Finally, I must extend special thanks to my wife, Norma, without whose patience and understanding this thesis could not have been completed.

I. INTRODUCTION

The purpose of this thesis was to investigate the feasibility of using microcomputers (such as the Intel® 8080) as Mode I block-by-block AUTODIN tributaries. Before embarking on the feasibility study, the AUTODIN was studied carefully to ensure that the problem was completely understood. Chapter II examines this background information, giving an overview and a functional description of the AUTODIN. In addition, the reasons for investigating microcomputers as potential AUTODIN tributary stations are explored.

Difficulty was encountered in understanding all ramifications of the AUTODIN protocol. As a consequence, the protocol was described in terms of a receive machine and a transmit machine, which are described in Chapter III. A step-by-step description of the software design of an AUTODIN test program is given in Chapter IV. Careful definition of the problem, understanding the hardware environment, and using the top-down, modular approach are the points emphasized.

® Intel and Intellec are registered trademarks of the Intel Corporation.

Chapter V describes the results of feasibility testing where the correctness of the AUTODIN test program was verified and timing tests demonstrated that the 8080 CPU could function as an AUTODIN tributary at modulation rates exceeding 2400 baud. Finally, Chapter VI summarizes the conclusions and recommendations of this thesis.

II. BACKGROUND

The Defense Communications System (DCS) consists of three major subsystems: the Automatic Voice Network (AUTOVON), the Automatic Secure Voice Communications Network (AUTOSEVOCOM), and the Automatic Digital Network (AUTODIN). The first two subsystems provide nonsecure and secure voice communications, while the third subsystem, AUTODIN, provides a secure record communication capability. This thesis is concerned with the AUTODIN.

A. AUTODIN OVERVIEW

The AUTODIN functions as a single, integrated, worldwide, high-speed, computer-controlled, general purpose communications network which provides record communications service to the Department of Defense (DOD) and other Federal Government Agencies, such as the Department of State. In addition to providing record communications service via various media (such as printed page, magnetic tape, Hollerith cards, etc.), the AUTODIN is also secure and fully automatic. It was designed, engineered, and programmed to provide responsive and continuous operation, minimal loss of service, and no loss of message traffic.

The AUTODIN is a network which consists of 17 Automatic Switching Centers (ASC's) and numerous tributary stations. Of the 17 ASC's, nine are leased and are located in the continental United States and Hawaii. The remaining eight ASC's are government owned and are located in Europe, the Pacific, and Alaska. Each ASC may have up to 300 tributary stations connected to it. This network of ASC's and tributary stations is able to provide responsive communications by use of a system of four message precedence levels: flash, immediate, priority, and routine. By requiring users of the AUTODIN to curtail their use of the higher precedence levels, and by programming the AUTODIN to handle all message traffic on a precedence basis, it is possible for flash precedence level messages to be switched and transmitted around the world in a matter of a few minutes. This capability for rapid communications greatly enhances the effectiveness of the defense establishment of the United States.

A flash level message interrupts all messages of precedence level immediate or less. Precedence level immediate messages are processed before priority or below level messages; however, the lower precedence level messages are not interrupted. Similarly, priority level messages are processed before routine level messages (without interruption of the routine level messages). By proper selection

of precedence levels, users of the AUTODIN can control the speed at which their messages are propagated through the system, with a lower limit of one to three minutes for flash messages and an upper limit of one to two hours for routine messages.

In addition to the variability of speed of transmission provided by the precedence system, there are two other properties of the AUTODIN which greatly enhance its usefulness. First, there is the capability for multiple addressing. The originator of an AUTODIN message may specify that the message go to one, two or hundreds of addressees. This can be accomplished in two ways: by enumerating the addressees, or (if the addressees are grouped together often) by use of collective addresses. The second additional property of the AUTODIN which enhances its usefulness is the ability to use various media for record communications. For example, the originator of a lengthy supply message might transmit the message from magnetic tape and the message could be received as cards on a card punch at the receiving communication center. Conversely, a small communication center without a card capability could transmit logistical data from paper tape and have it punched as cards at the receiving communication center, thus eliminating the need for keypunching the data at the logistical center.

With such properties as variable speed of transmission, selectable media input and output, and multiple addressing, the AUTODIN has provided flexible, responsive, and reliable record communications for over a decade. In order to understand it more fully, it is necessary to examine it on a more technical and detailed level.

B. FUNCTIONAL DESCRIPTION OF AUTODIN

The AUTODIN is a digital network consisting of ASC's and tributary stations with interconnecting communications channels. Both synchronous and asynchronous operation are employed within the AUTODIN; however, asynchronous operation is permitted only on tributary channels, whereas synchronous operation is permitted on both interswitch trunks and tributary channels. For synchronous operation, the AUTODIN will process information at modulation rates of 75, 150, 300, 600, 1200, 2400, and 4800 baud. For asynchronous operation, modulation rates of 75, 150, and 300 baud are permitted. All synchronous AUTODIN communications channels use the American Standard Code for Information Interchange (ASCII). The basic unit for information transfer in AUTODIN is the line block, several of which are shown in Figure 1.

1. Modes of Operation

There are five modes of operation within the AUTODIN. These are Mode I, which is duplex, synchronous operation with

automatic error and channel controls which allow independent and simultaneous two-way operation; Mode II, which is duplex, asynchronous operation allowing simultaneous two-way operation without automatic error and channel controls; Mode III, which is duplex, synchronous operation with automatic error and channel controls (but with one-way information transfer and the return direction used solely for error control and channel coordination responses); Mode IV, which is a unidirectional synchronous operation which can send only or receive only and does not have automatic error control; and Mode V, which permits duplex asynchronous operation and allows simultaneous and independent two-way transmission but which performs only limited channel coordination and display functions.

From the above descriptions, it should be evident that Mode I AUTODIN is the most efficient and hence most desirable type of AUTODIN. All of the asynchronous modes are limited to modulation rates of 300 baud or less. Thus, for medium or high speed data transfer rates, the synchronous modes (Mode I or Mode III) must be used. Mode III contains an inherent disadvantage in that information transfer (or message transmission) is limited to one direction at a time. Thus, only Mode I AUTODIN offers both high-speed operation and simultaneous and independent two-way transmission of

information. This thesis deals solely with Mode I block-by-block operation. All subsequent discussion of AUTODIN assumes Mode I block-by-block operation. The difference between block-by-block and continuous operation will be discussed in Section II.B.5 of this thesis.

2. Synchronous Idle Pattern

In Mode I AUTODIN operation, whenever information is not being transmitted, synchronous idle pattern must be transmitted at the designated modulation rate. Synchronous idle pattern is an even parity character which is equal to the number 96 hexadecimal (or 10010110 binary). Since synchronous idle is transmitted whenever information is not being sent, the receive side of the AUTODIN logic uses synchronous idle pattern to determine whether or not it is in synchronization. At initialization, the Mode I AUTODIN receiver attempts to detect the synchronous idle character (SYN). After the first SYN is detected, the next three characters are checked for the SYN pattern. If the following three characters are SYN, then the receiver considers itself to be in character frame (or synchronized); otherwise, it repeats the above process, repeatedly attempting to achieve character frame. An AUTODIN transmitter may transmit information only if its receiver is in character frame. Likewise,

an AUTODIN receiver may process incoming information only if it is in character frame.

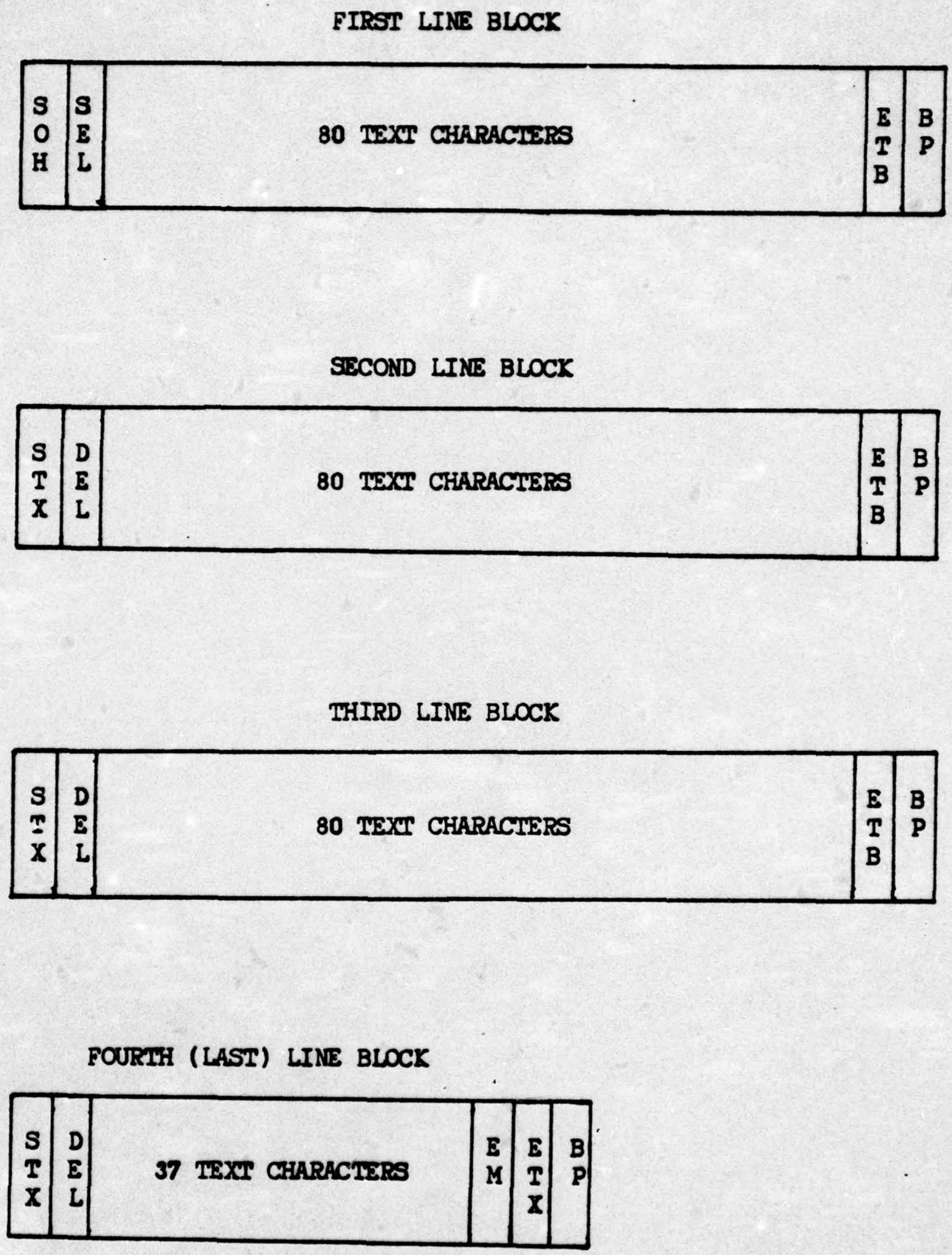
3. Line Block Format

The basic unit for information transfer in AUTODIN is the line block. It may be thought of as a package of information. A typical sequence of events for an ASC transmitting to a tributary station under Mode I operation might be as follows: The ASC sends synchronous idle pattern to the tributary station. The tributary receiver recognizes the synchronous idle pattern and considers itself in character frame. Since Mode I AUTODIN is duplex, the same process takes place (simultaneously and independently) in the opposite direction: the tributary transmitter achieves synchronization with the ASC receiver. Once synchronization has been achieved, it is possible to transmit information in the form of line blocks or "packages" of information. If, for instance, the ASC were transmitting to the tributary, the ASC would send the first line block. If the tributary station received the line block without error, it would reply with an acknowledgement, and the ASC would be free to send a subsequent line block. However, if any error were present in the line block, the tributary would reply with a negative acknowledgement (NAK), and the ASC would retransmit the first line block. In this manner, information is transmitted

in either direction or both directions with channel control and error detection. It should be kept in mind that in Mode I AUTODIN, simultaneous and independent information transfer can occur in both directions. In order to understand more fully the AUTODIN communications protocol, it is necessary to examine the line block structure and associated control characters in detail.

Consider an AUTODIN message which contains 277 text characters or bytes of information. It would be transmitted as four line blocks, the first three of which would contain 80 bytes of information while the fourth would contain 37 bytes of information. Figure 1 shows the line block structure of such a message.

The first character of the first line block of every AUTODIN message is the Start of Heading (SOH) framing character. It is an even parity character which signals the beginning of a new message, and it is always followed by the Select (SEL) framing character. This sequence cannot be split by any other character. The SEL character is an even parity framing character which is always the second framing character of the first line block of every AUTODIN message. Unlike the SOH framing character which is always the same, the SEL character may be one of several alphabetic characters. These alphabetic characters correspond to the



LINE BLOCK STRUCTURE OF AN AUTODIN MESSAGE
CONTAINING 277 TEXT CHARACTERS

FIGURE 1.

various Language Media Format (LMF) indicators but are coded by a different set of characters, according to reference 10. The LMF characters, which appear in the message as it enters and leaves the system, correspond one for one with the SEL characters, which appear in the first line block of the message while it is inside the network. The translation from LMF character to SEL character and back must be accomplished by the network interfaces (tributaries). For example, if an AUTODIN message were narrative in nature, and the originator desired that the addressee of the message receive a printed page version of the message, then the originator would use the LMF indicators "TT." The second "T" would indicate that output on a line printer (or similar device) was desired. This "T" would be translated into the SEL character "H" by the transmitter. Thus, the receiver at an AUTODIN tributary station which received an SOH followed immediately by an even parity "H" would interpret this to mean that the incoming message was to be printed on the line printer. The purpose, therefore, of the SEL character is simply to select the output device at the receiving tributary station. An LMF "C" (meaning card output) would be translated into a "D" SEL character which would cause output on a card punch at the receiving tributary. Reference 2 contains a complete list of SEL and LMF characters.

Following the SOH and SEL framing characters are the first 80 text characters of the AUTODIN message. These characters are transmitted with odd parity. That is, the first seven bits correspond to the American Standard Code for Information Interchange (ASCII) and the eighth bit is either a one or zero such that the total number of ones in the eight-bit byte are odd. The next-to-last character in the first line block of the example in Figure 1 is the End of Transmission Block (ETB) framing character. In fact, ETB is always the third framing character of every line block except the last block of the message. Like all other framing characters, it is an even parity character. The ETB character is immediately followed by the Block Parity (BP) character. No character of any kind may be inserted between ETB and BP. Block Parity is the last framing character of every AUTODIN line block. It may be either odd or even in parity because it is formed by the binary addition without carry (sum modulo 256) of all bytes in the line block. In this way BP serves to check the correctness of received line blocks by detecting single errors.

The second line block of the example message begins with the Start of Text (STX) framing characters. STX is the first framing character of every line block except the first line block which is started with the SOH framing

character. STX is an even parity character which is always followed immediately by a Delete (DEL) framing character, which is also even in parity. The DEL character is the second framing character of every line block except the first one which has an SEL in the second position. The DEL character is used only on links between ASC's and tributary stations. On interswitch trunks between ASC's, the DEL is replaced with a Security (SEC) framing character which is used by the ASC's for the routing of classified and unclassified message traffic.

The remainder of the second line block is the same as the first line block -- 80 text characters followed by the ETB and BP characters. In fact, all subsequent line blocks are the same (STX, DEL, 80 text characters, ETB, and BP) except for the last line block. The last line block begins with STX and DEL framing characters; however, these are followed by 37 text characters and three framing characters. The first of these framing characters is the End of Medium (EM). This even parity character is used to signal the end of an AUTODIN message. It is followed by the End of Text (ETX) framing character (even in parity) and the BP framing character. The BP character is formed as previously described except that it is computed on the 37 text characters and the EM character instead of the 80 text characters as in line blocks one, two, and three.

The line block structure is built and transmitted by the transmit logic of an AUTODIN ASC or tributary. Analogously, the receive logic portion of an AUTODIN ASC or tributary expects to receive information in this line block structure. Now that this structure has been explained, it is possible to discuss the AUTODIN protocol and its associated control characters.

4. Control Characters

In order to provide for channel coordination, control characters are required. Control characters are even parity characters which are always transmitted as contiguous pairs. Six of the most important ones are described below:

(1) Acknowledge Number One (ACK1).

ACK1 is sent by an ASC or tributary to signal the distant transmitter that a line block has been received correctly. ACK1 is the answer to the first line block sent after power-up, or to the first line block received after a message has been cancelled. Thereafter, ACK1 is used alternately with ACK2 to indicate correctly received line blocks.

(2) Acknowledge Number Two (ACK2)

ACK2 is sent as a reply to indicate the correct reception of a line block after a line block has been acknowledged with ACK1. For example, if line block one is received correctly and an ACK1 is sent in reply, then when line block

two is received (correctly), an ACK2 is sent in reply. The sequence of alternate ACK1's and ACK2's is not interrupted between messages; that is, if the answer to last line block of a message was ACK1, then the answer to the first line block of the next message will be ACK2.

(3) Negative Acknowledge (NAK)

Tributaries and ASC's use NAK to signal that a line block has been received with an error in it. NAK is sent after the end of the erroneous line block is received, not at the time the error is detected. Whenever an NAK is received, the transmitting station will retransmit the complete line block to which the NAK applies.

(4) Reject Your Message (RM)

RM's are sent as replies to line blocks. Only an ASC can send an RM, which is sent to the transmitting tributary to signal that there is a defect in the message which cannot be rectified by retransmission of the line block.

(5) Wait Before Transmitting (WBT)

WBT is sent by either an ASC or tributary station in response to a properly framed line block to inform the distant transmitter that the local receiver can no longer accept line blocks. The eventual response to the line block in question may be an ACK1, ACK2, or even NAK; however,

while WBT is being received (and until an ACK or NAK is received), the transmitting station may send only control characters or synchronous idle pattern (SYN).

(6) Reply (REP)

An ASC or tributary station transmitter will use the REP to direct the distant receiver to send its last response or current (updated) response such as ACK1, ACK2, NAK, RM, or WBT. Each transmitter must be equipped with a variable timer hereafter referred to as the answer timer. At the end of each line block transmitted, the answer timer is initialized. When the answer timer expires an REP will be sent if an answer has not been received or if a WBT has been received. Each time an REP is sent the answer timer will be reinitialized. Whenever an ACK1, ACK2, NAK, or RM is received, the answer timer will be stopped. The duration of the answer timer is a function of modulation rate, communication path delays, delays in modems, and receiver response delays. The answer timer duration is determined by adding together all the delays for an expected round trip delay time plus a safety margin. Thus, the answer timer delay is equal to slightly more than the time to receive an expected answer (ACK1, ACK2, NAK, etc.) to a line block or REP. Typical answer timer settings are 3 seconds for 75 to 600 baud circuits, 0.5 seconds for 1200 baud circuits

and 0.25 seconds for 2400 baud circuits. If REP is sent three times in succession without receiving an appropriate reply, an alarm will be sounded.

(7) Cancel (CAN)

CAN is sent by a transmitting station to signal the distant receiver to cancel or discard the current message. The CAN may be initiated manually, automatically by the transmitter upon an incorrectable error condition, or automatically by the receiver whenever an RM is received as the response to a line block.

The aforementioned seven control characters permit channel coordination such that erroneous line blocks are retransmitted, correct line blocks are acknowledged, and, whenever circuit degradation occurs, alarms are activated which bring the requisite human intervention. The next section provides examples which will demonstrate the inter-operative relationship between line blocks, framing characters, and control characters.

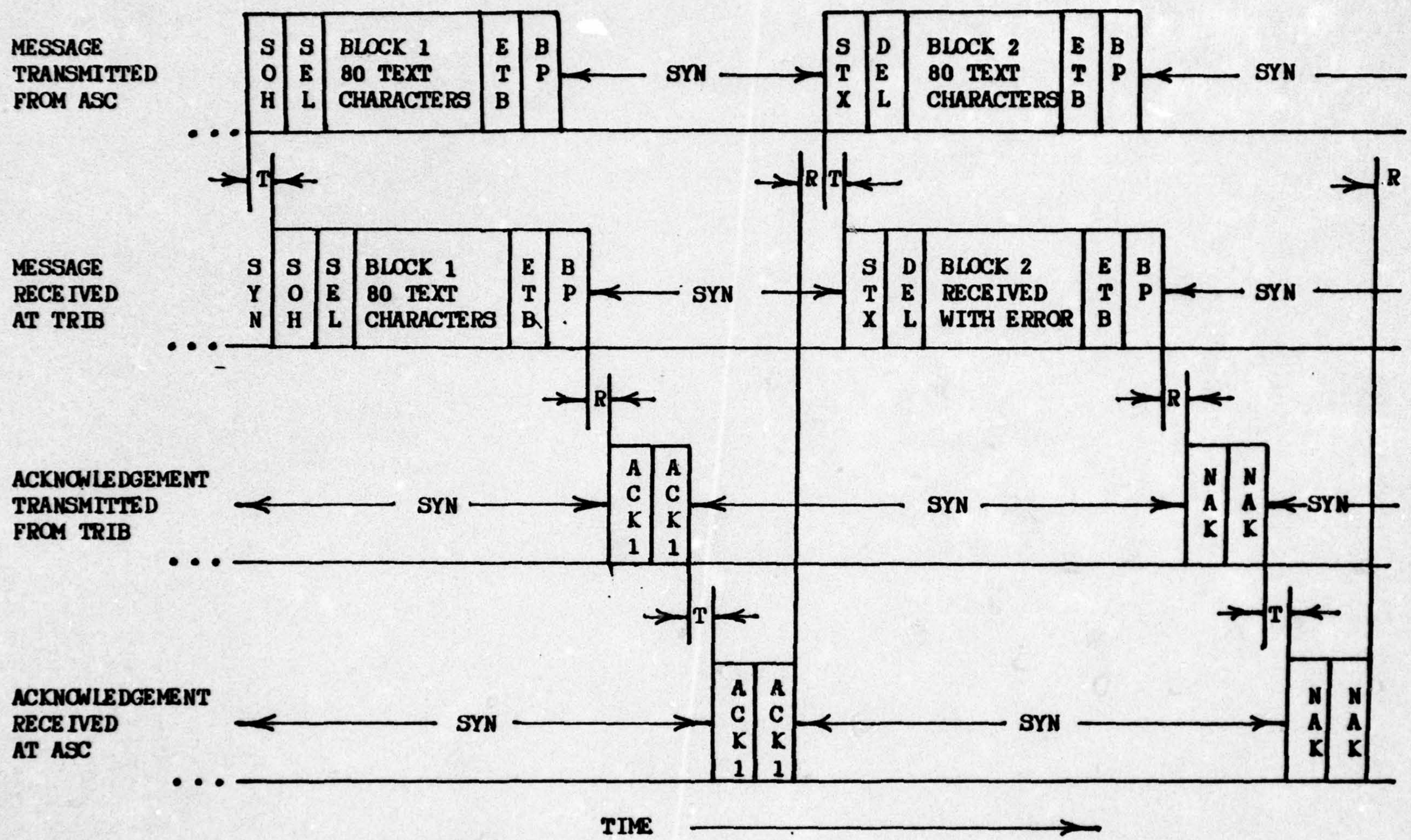
5. <u>An Analysis of Block-By-Block Operation</u>

Within Mode I AUTODIN there are two types of operation: block-by-block and continuous. Under block-by-block operation, a transmitting station sends one line block and does not send a subsequent line block until an ACK1 or ACK2 is received. Under continuous operation, one line block is

sent, then a second one. When continuous operation is working properly, the ACK for the first line block will be received while the second is being transmitted. There is no difference between block-by-block and continuous mode for an AUTODIN receiver and only a trivial change in buffering for an AUTODIN transmitter. This thesis deals only with block-by-block operation.

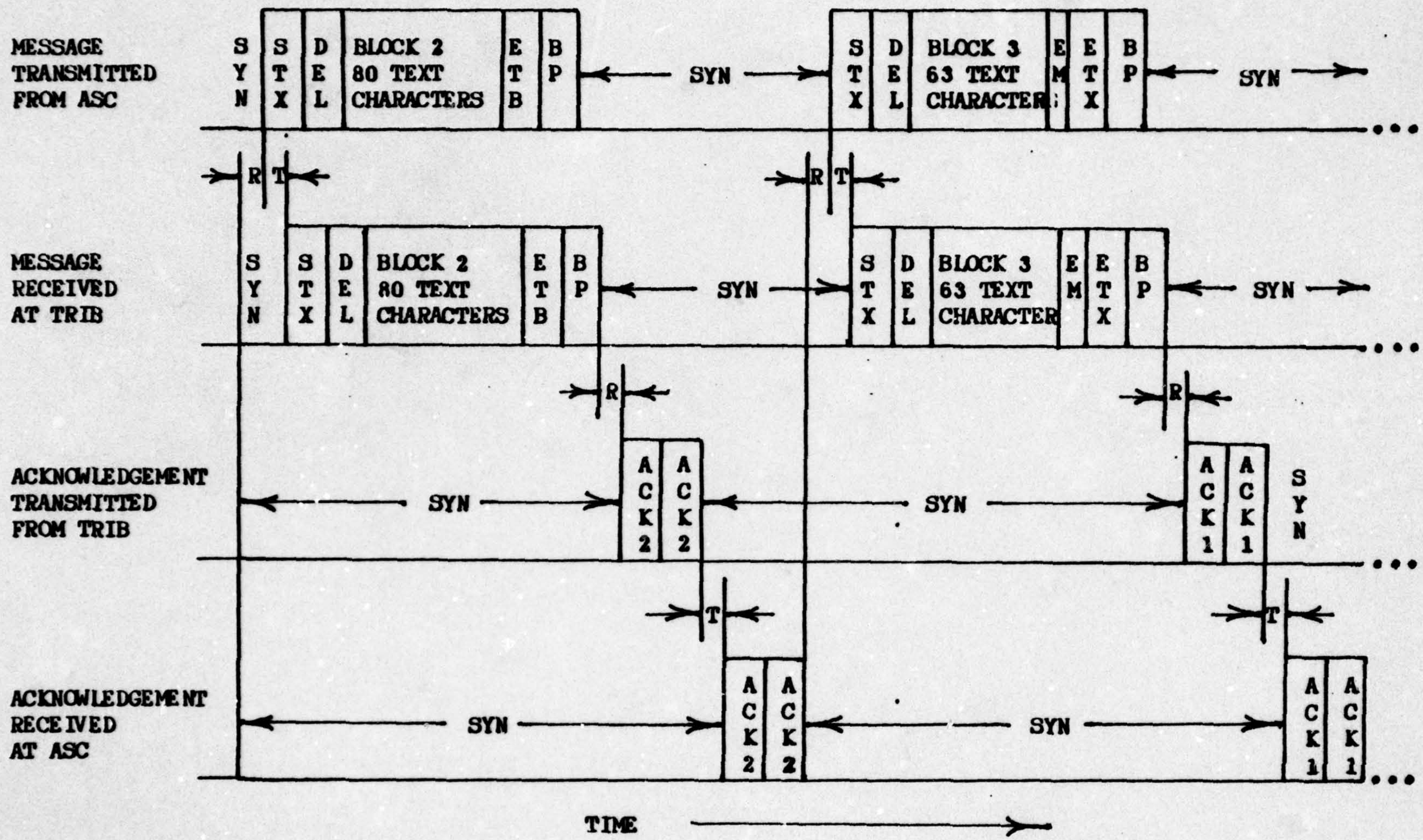
Figure 2 illustrates the AUTODIN protocol: the transmission of data in line block format, the channel coordination obtained from the control characters, the synchronous idle pattern between line blocks and the transmission and response delays involved. The message being transmitted in the example of Figure 2 contains 223 text (or informational) characters. This requires two full-size line blocks of 80 text characters each and a third line block of 63 text characters. In this example, the information transfer is in one direction with the ASC transmitting and the tributary receiving. It will be instructive to trace through Figure 2 from left to right, noting that moving from left to right is analogous to moving forward in time.

Line block one with SOH and SEL for beginning framing characters is transmitted from the ASC and is received at the tributary after a transmission time delay (denoted by "T").



BLOCK-BY-BLOCK OPERATION IN ONE DIRECTION WITH ASC TRANSMITTING AND TRIBUTARY RECEIVING

FIGURE 2a.



BLOCK-BY-BLOCK OPERATION IN ONE DIRECTION WITH ASC TRANSMITTING AND TRIBUTARY RECEIVING

FIGURE 2b.

After the entire line block is correctly received, and after a response delay time (denoted by "R"), the tributary sends two contiguous ACK1's. These are received back at the ASC a transmission time (T) later. The ASC then transmits the second line block of the message; however, this time there is an error in transmission. Consequently, the tributary sends two contiguous NAK's. When these NAK's are received at the ASC, the second line block of the message is retransmitted. This time it is correctly received by the tributary, which sends the appropriate ACK2. Finally, the last line block of the message, which is a short line block containing 63 text characters (marked at the end of text by an EM character), is transmitted. The tributary acknowledges receipt of the last line block with an ACK1, which illustrates the alternation of ACK1's and ACK2's.

From the above descriptions, the reader should have a general idea of how Mode I block-by-block AUTODIN functions. It is merely the transmission, reception, and acknowledgement of line blocks which contain the information to be communicated.

C. REASONS FOR INVESTIGATING MICROCOMPUTERS FOR AUTODIN APPLICATIONS

Reference 1 states that various computers have been approved and certified by the Defense Communications Agency

(DCA) for use as AUTODIN tributary stations. These computers are:

1. IBM 360 Series.
2. RCA SPECTRA 70 Series.
3. Univac DCT 9000 Series.
4. Control Data Corporation CD1700 Series.
5. SOROBAN DST (Mohawk Data Science Corporation).
6. Honeywell 200 Series.
7. ITT World ADX 9300.
8. Burroughs 3500 Series.

Hundreds of the above machines have been installed around the world to provide AUTODIN service to the far-flung units of the Department of Defense. The Naval Telecommunications Center, Monterey, California, is a typical tributary station. It is a 1200 baud Mode I tributary which consists of a Univac DCT 9000 computer with two magnetic tape drives, a card reader, a card punch, a paper tape reader, a line printer, and a communication interface unit. The annual cost to the government to provide this equipment is $67,824.00 per year for equipment leasing and $13,512.00 for on-call maintenance support. Thus, for equipment alone, over $80,000.00 per year must be spent on this tributary station, and this is not an atypical amount. The Communication Center of the Third Force Service Regiment, Fleet Marine Force Pacific, located on the island of Okinawa, costs a similar amount for the same capability: a 1200 baud, Mode I AUTODIN tributary. In this case the equipment is an IBM 360/20 with equivalent peripheral equipment.

In reviewing the above equipment costs, two questions immediately come into mind: First, are such relatively expensive and powerful computers needed for AUTODIN tributary applications? Second, can inexpensive microcomputers function as AUTODIN tributaries? If microcomputers can be programmed to serve as AUTODIN tributary stations, then it is possible to replace the more expensive, powerful machines presently being used and save millions of dollars each year. In addition, since microcomputers are smaller, lighter, and more rugged than the aforementioned large computers, the potential use of microcomputers as AUTODIN tributaries in tactical and mobile situations could greatly improve the record communication capabilities of deployed combat units. In short, greatly reduced costs and expanded AUTODIN service in tactical situations are two potential benefits to be realized if microcomputers are capable of functioning as AUTODIN tributaries. For this reason, the central question of this thesis is: can a microcomputer function as an AUTODIN tributary? If so, how <u>fast</u> can it process information?

III. MAKING THE AUTODIN PROTOCOL MORE UNDERSTANDABLE

Before designing and writing a computer program which would demonstrate the feasibility of using a microcomputer as an AUTODIN tributary, it was necessary to understand completely all of the details of the AUTODIN protocol for Mode I block-by-block operation.

A. DIFFICULTIES IN UNDERSTANDING THE AUTODIN PROTOCOL

Although reference 2 is a very comprehensive and detailed document, it is difficult to use in gaining a complete and precise understanding of the AUTODIN protocol. The major obstacle which prevented an easy understanding of the protocol is the limitation of short-term human memory: it was impossible for the author to digest reference 2 from cover to cover and then suddenly realize and understand the exceedingly complex AUTODIN protocol. The problem was that reference 2 failed to approach the problem of describing AUTODIN from the top down. In other words, instead of giving an overview of AUTODIN and then explaining it in levels of increasing detail, reference 2 appeared to approach the problem from the inside out, a method which was not suitable for rapid and easy understanding of the protocol. This

contention is reinforced by the following example. In 1975, after over a decade of AUTODIN service, the Univac DCT 9000 computer at the Naval Telecommunication Center, Monterey, California, went into the machine halt condition as the result of a software bug which surfaced while an AUTODIN message was being transmitted. Reference 10 specifies that AUTODIN messages shall be terminated by eight line-feeds followed by four N's. However, the DCT 9000, a computer which is sanctioned for AUTODIN use by DCA, interpreted the presence of four contiguous N's in an encoded weather message as the end of message indicator. No line-feeds were involved. This lack of precision in describing the AUTODIN protocol leads to ambiguities which can cause mistakes in programming. The process of describing a complex, detailed protocol in this manner is analogous to describing a building to a blind man brick by brick without first giving a description of the shape, size, and purpose of the building.

For example, when reading reference 2, the author came across the fact that all control characters are transmitted in contiguous pairs. The question then arose as to what the AUTODIN receive logic must do in the event only one control character is received. Should the receiver ignore the character? Should it act upon the character as though it were a valid two-character control sequence? At first, the

author thought that there was an ambiguity on this point; however, the answer was finally found buried in the details of reference 2: the receiver logic ignores single control characters; it only acts upon contiguous pairs of control characters.

This example and others like it served to illustrate the inadequacies of reference 2. What was needed was an overview of the protocol -- some method of describing the inter-operability of all the facets of the protocol. The flowcharts of reference 2 failed to provide an overview of the protocol and also failed to provide enough precision to cover all contingencies. In other words, a better method of describing the AUTODIN protocol was needed. This better method was first used by Renninger in reference 12.

Renninger described the AUTODIN protocol in terms of two state transition diagrams: a receiver and a transmitter. Indeed, throughout reference 2 there are numerous references made to a receiver and a transmitter, but the reader is never told exactly what they are. From studying the work of Renninger, it became clear to the author what the AUTODIN receiver and transmitter were: they were transition state machines which had starting states and which were driven from state to state. Each incoming or received byte represented a potential state transition for the receive machine;

likewise, each byte to be transmitted represented a potential state transition for the transmit machine. The actual transitions made and actions taken depend upon these inputs to the receiver and transmitter and upon the condition of numerous flags which contain detailed information on the overall state of each machine.

The term transition state machine is used here to denote a machine which is derived from and closely related to a finite state machine. The major difference is that while a finite state machine uses only states to define its logic, a transition state machine uses both flags and states. This somewhat more informal method of describing a logical process has two advantages over the finite state machine model. The first of these is that it permits designers to concentrate on the most important states and the second advantage is that the problem can be reduced to an understandable and readable form. The AUTODIN receive and transmit machines are described in more detail in the sections that follow.

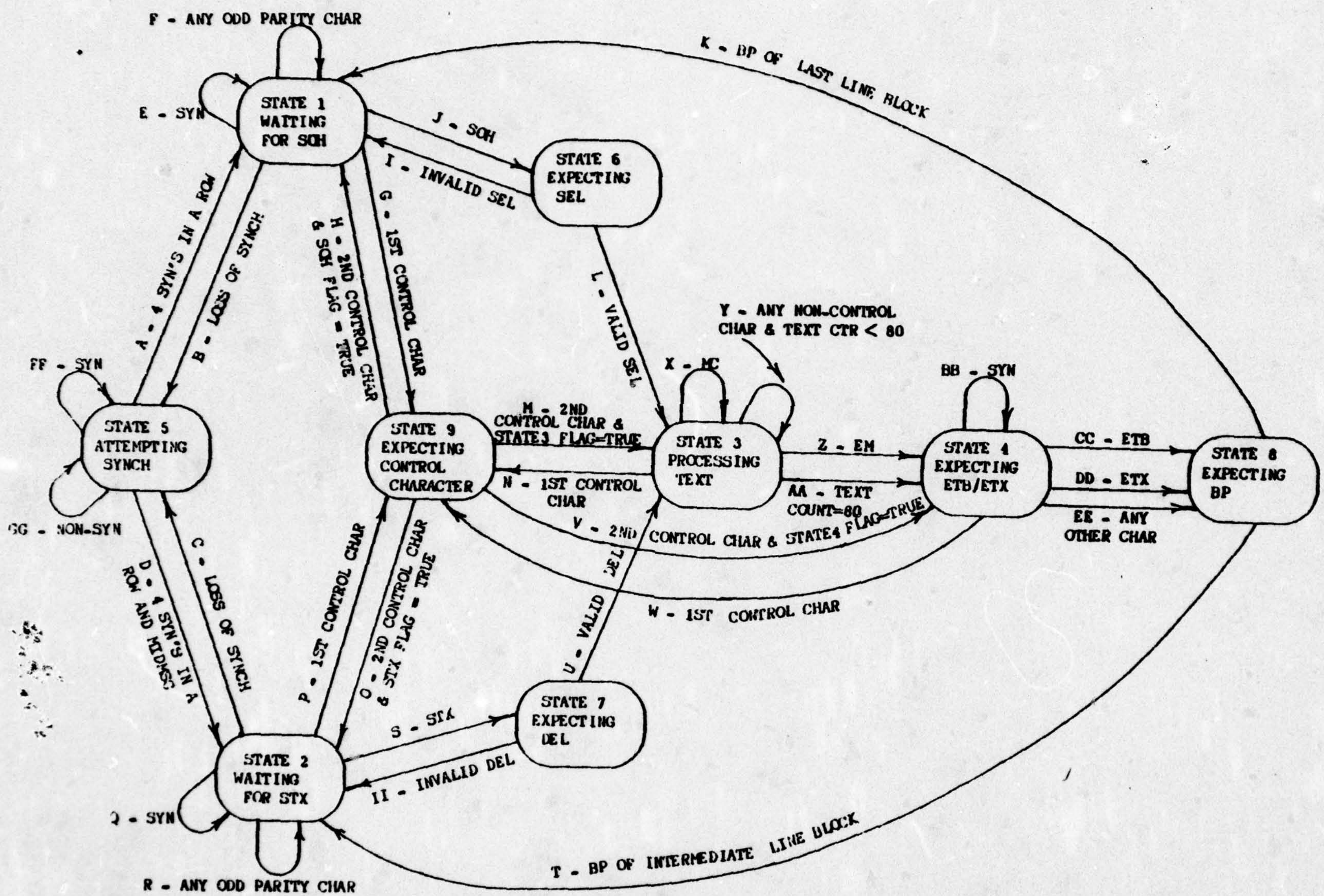
B. THE AUTODIN RECEIVE MACHINE

Renninger described the Mode I AUTODIN receive protocol as a 17-state machine. Here the receive protocol is specified as a nine-state machine. The reason that the protocol can be specified here with eight fewer states is that this

version of the protocol uses more condition flags than Renninger's model, but fewer states. Thus, the two machines are logically equivalent with the following exception: Renninger's machine is for continuous Mode I AUTODIN whereas this machine is for block-by-block Mode I AUTODIN.

Figure 3 depicts the AUTODIN receiver in the form of a nine-state transition diagram. The states are numbered, and the transition paths between states are labeled with letters. A description of each state transition is given in Table I. Each incoming byte or message character corresponds to a transition line on the state diagram, with some transitions beginning and ending in the same state.

It is felt that the state diagram of Figure 3 is a superior method of specifying the control logic of the AUTODIN receiver. It is superior to the flow charts and explanatory text of reference 2 because it utilizes the concept of a finite state machine in its graphical representation to completely specify on one page the receiver protocol. Obviously this is better than scores of pages of text and flowcharts.



THE AUTODIN RECEIVER STATE DIAGRAM
Figure 3

TABLE I

AUTODIN RECEIVER STATE TRANSITION DESCRIPTIONS

Transition	Description
A	Synchronization achieved between messages. Four SYN's received in a row and mid-message flag = false.
B	Loss of synchronization between messages. Receiver timer has expired without 4 SYN's being received and mid-message flag = false.
C	Loss of synchronization between line blocks. Receiver timer has expired without 4 SYN's being received and mid-message flag = true.
D	Synchronization achieved between line blocks. Four SYN's received in a row and mid-message flag = true.
E	SYN character received. Increment syn-counter. If syn-counter = 4 then reset receiver timer and set SYN-COUNTER = 0.
F	Any odd parity character received. Set syn-counter = 0. Check to see if receive timer is expired.
G	First character of a two-character control sequence received. Set SOH flag = true.
H	Second character of a two-character control sequence received and SOH flag = true.
I	Invalid SEL character received.
J	SOH received.
K	BP of last line block in message received (last line block because mid-message = false.

Transition	Description
L	Correct SEL received. Set mid-message flag = true.
M	Second character of a two-character control sequence received and text flag = true.
N	First character of a two-character control sequence received. Set text flag = true.
O	Second character of a two-character control sequence received and STX flag = true.
P	First character of a two-character control sequence received. Set STX flag = true.
Q	SYN character received. Increment syn-counter. If syn-counter = 4 then reset receive timer and set syn-counter = 0.
R	Any odd parity character received. Set syn-counter = 0. Check to see if receive timer is expired.
S	STX received.
T	BP of intermediate line block in message received (intermediate because mid-message flag = true).
U	Correct DEL received. Set mid-message flag = true.
V	Second character of a two-character control sequence and ETB flag = true.
W	First character of a two-character control sequence. Set ETB flag = true.
X	MC received. Set MC flag = true.
Y	Any non-control or non-framing character received and text counter is less than 80. Increment text counter.

Transition	Description
Z	EM received. Set mid-message flag = false.
AA	Text counter = 80.
BB	SYN received.
CC	ETB received.
DD	ETX received.
EE	Any character other than ETX or ETB received. Set error flag = true.
FF	SYN received. Increment syn-counter.
GG	Any character other than SYN received. Set syn-counter = 0.

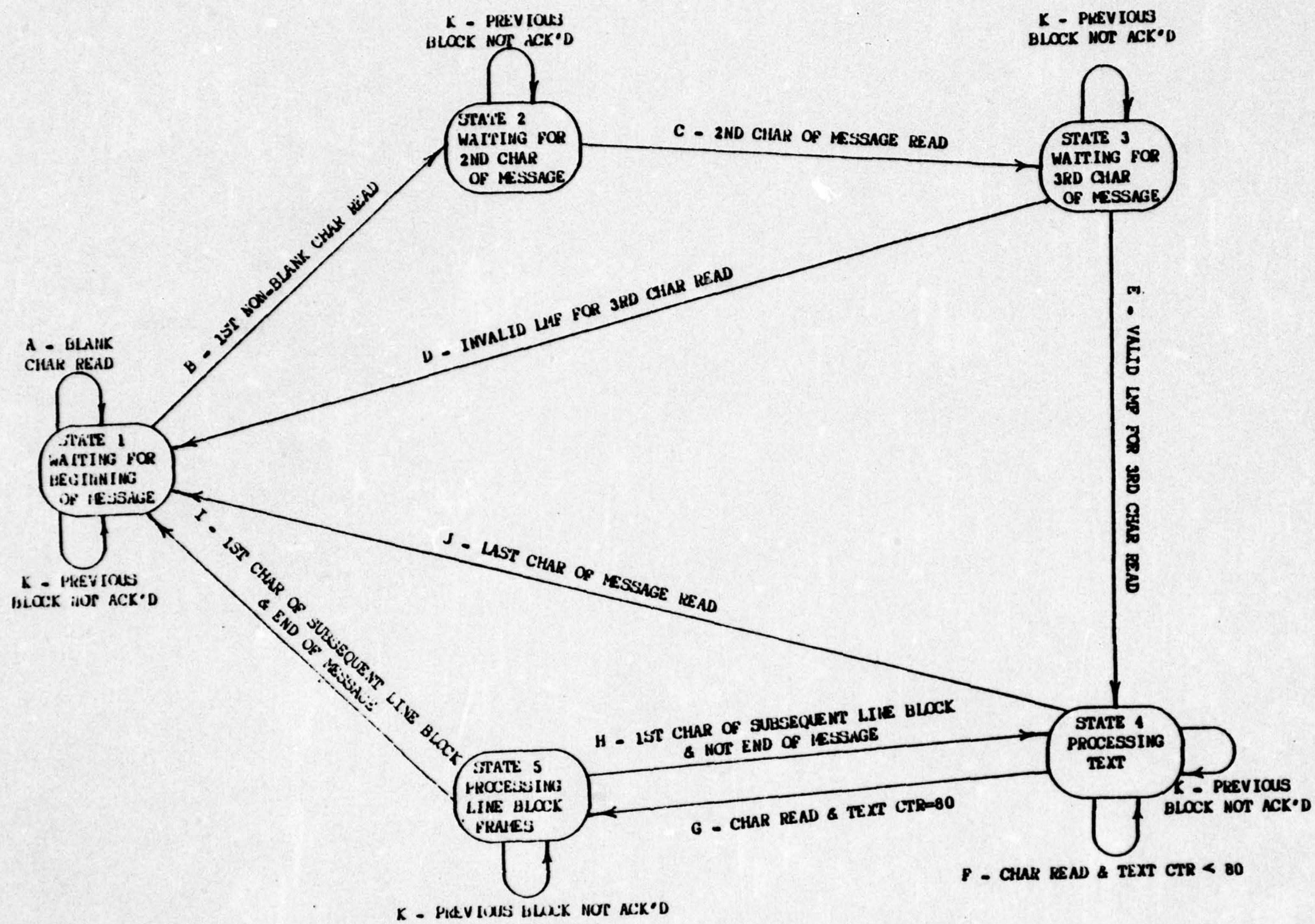
C. THE AUTODIN TRANSMIT MACHINE

The design of the AUTODIN transmit machine is very similar to that of the receive machine except, of course, that the purpose of the transmit machine is to read and transmit characters of text while the purpose of the receive machine is to receive text. Again, Renninger's transmitter consists of nine states whereas the author's machine consists of five states. The reason for fewer states is the same as for the difference in receiver states: fewer states, more condition flags. Finally, this transmit machine and Renninger's differ in that the former is for block-by-block operation and the latter for continuous operation. Otherwise, they are functionally equivalent.

Figure 4 depicts the five-state AUTODIN transmitter as a state transition diagram and Table II gives a description of each of the transitions. Each outgoing byte or text character which is read by the transmit machine corresponds to a line on the state transition diagram. These outgoing bytes can cause transitions from state to state or from a state back to the same state. Again, it is felt that the state diagram method of specifying a communication protocol is far superior to the method used in reference 2.

It should be pointed out that the receive and transmit machines, while specifying the AUTODIN protocol, do not

completely specify all of the details required for implementation on an actual computer. The actual implementation of the receive and transmit machines is the subject of the next section.



THE AUTODIN TRANSMITTER STATE DIAGRAM
Figure 4

TABLE II

AUTODIN TRANSMITTER STATE TRANSITIONS

Transition	Description
A	Blank character read. Blanks are used as leader on paper tape messages and are read and discarded by the transmitter.
B	First non-blank character of a message read. Place character in outgoing line block.
C	Second non-blank character of a message read. Place character in outgoing line block.
D	Third non-blank character of a message read and table lookup indicates invalid LMF character. Cancel the message.
E	Third non-blank character of a message read and table lookup indicates valid LMF character. Place SEL character in outgoing line block and set text counter = 4.
F	Character read and text counter is less than 80. Increment text counter. Place character in outgoing line block.
G	Character read and text counter = 80. Set text counter = 0 and place character in outgoing line block.
H	First character of subsequent line block and end-of-message = false. Set text counter = 2. Place character in outgoing line block.

Transition	Description
I	First character of subsequent line block and end-of-message = true. Place character in outgoing line block.
J	Character read and end-of-message = true. Place character in outgoing line block and set text counter = 0.

IV. SOFTWARE DESIGN AND IMPLEMENTATION

The central issue of this thesis is whether or not a microcomputer can function as a Mode I AUTODIN tributary station and, if it can, at what baud rate can it so function? In order to answer this question it became clear from the very beginning that it would be necessary to develop a computer program which would function as an AUTODIN tributary station. In this way, it would be possible to determine whether or not a microcomputer could perform all of those functions associated with an AUTODIN tributary. If this test proved to be successful, that is, if the microcomputer could perform the necessary tributary functions, then a timing test could be devised which could measure the rate at which the microcomputer could function as an AUTODIN tributary. Consequently, a major portion of the effort expended in this thesis was spent on designing, developing, implementing, testing, and timing a computer program which enabled a microcomputer to function as an AUTODIN tributary station. It should be emphasized, however, that the purpose of the AUTODIN test program is to demonstrate feasibility. It was not designed for actual AUTODIN use.

A. DEFINITION OF THE PROBLEM

The first step in the top-down approach to software development is to define completely the problem to be solved. In this case, the problem was to write a computer program which would enable a microcomputer to function as an AUTODIN tributary station. In addition, the program was to have the property that it could be timed to determine the rate at which it could process AUTODIN messages. This statement defined the problem at its highest level or most general form.

The problem at hand was then taken to the next level of detail. It was determined that the microcomputer, in order to function as an AUTODIN tributary, should be able to interface with the receive side of a communication channel, perform the receive functions of the AUTODIN protocol, and pass the text obtained from the receive channel to a writer device such as a line printer, card punch, or magnetic tape drive. Simultaneously, the microcomputer must also read information from a reader device (such as a paper tape reader, card reader, or magnetic tape drive), put this information into line block format, and interface with the transmit side of a communication channel. In addition, there must be coordination between the transmit and receive functions to provide the full channel coordination and error

detection capability specified and required by the AUTODIN protocol.

The above paragraph represents an important step in the definition of any software problem. That step is specifying the operations which the program must perform. In many circles, this step (or the document which explains it) is called an operational specification. Appendix A is the operational specification for the AUTODIN test program to demonstrate the feasibility of using microcomputers in DCS AUTODIN applications. The reader will note that the operational specification was written in the future tense, since it was developed before the program.

In defining the problem to be solved, two accomplishments served to bring the problem into sharp focus. The first of these was the development of the operational specification of the program. The second was the development of the transmit and receive machine descriptions of the AUTODIN protocol. Indeed, putting the AUTODIN protocol into understandable form was the single most important aspect of defining the problem. The transmit and receive machine descriptions of the AUTODIN protocol appear to be hardware independent; however, many of the points discussed by the operational specification address hardware-dependent problems. For this reason (and in order to achieve an actual

implementation of the AUTODIN protocol) it was necessary to examine the hardware environment in which the program must reside.

B. THE HARDWARE ENVIRONMENT

The Intellec 8/Mod 80 microcomputer development system with its 8080 microprocessor CPU was chosen to develop and test the AUTODIN test program. The first reason for this choice was availability; however, many other reasons also existed. Among these were the wide use of 8080 CPU's (indeed, the 8080 has become an industry standard), the availability of software (such as high-level languages, debuggers, loaders, etc.) for program development and testing, and the ability to address up to 256 peripheral devices. In general, the 8080 is a single-chip, large-scale integrated (LSI) CPU which has 8 and 16-bit registers and can address up to 64K of main memory. References 4 and 5 provide more details on this subject.

The actual microcomputer used for development and testing of the AUTODIN test program was an Intellec 8 microcomputer with 8080 CPU, 16K of main memory, and two input/output (I/O) boards. The first I/O board was configured to permit interfacing with either a teletype or a cathode ray tube (CRT) terminal. The second I/O board was configured to work with

a Universal Asynchronous Receiver/Transmitter (UART). This hardware configuration obviously did not match the normal one found at a Mode I tributary station, which includes magnetic tape drives, card reader, card punch, paper tape reader, and line printer as well as a USART (as opposed to the UART available with the Intellec 8). In addition, in order to assure correctness of the AUTODIN test program, it was decided that tests with actual peripheral devices must be conducted. In order to conduct such tests, an equipment test configuration was developed.

First of all it was decided that the Intellec 8 could be tested back-to-back, with its transmit logic sending to its own receive logic to simulate the full duplex information transfer found on a Mode I communication channel between an ASC and tributary. In fact, as program development progressed, it became obvious that it made no difference whether or not the receiver was receiving information sent by itself (the same computer) or whether it was receiving information from a distant computer. The same was true for the transmitter. Only one minor logical difference became apparent: in using two machines, the receiver, at power-up, would attempt to achieve synchronization before permitting the transmitter to send anything. Obviously, if nothing was ever sent, then SYN characters could never be received. Consequently, for

the back-to-back configuration, a virtual bitstream was programmed into main memory, and part of the initialization of the program would insert SYN characters in this bitstream to achieve initial synchronization. Thereafter, the receive process would fetch bytes from this bitstream just as though it were interfacing with an actual USART. Conversely, the transmit process would insert bytes into this virtual bitstream just as though it were communicating with an actual USART. A side benefit of this method was that it eliminated use of the Intellec 8's UART. The UART was not used for two reasons. First, Mode I AUTODIN calls for synchronous vice asynchronous channel operation. Second, the UART available for testing was configured for seven-bit operation which precluded the use of eight-bit bytes. Eight-bit bytes with odd and even parity are mandatory for the AUTODIN logic. The use of the virtual bitstream concept solved both of these problems and did not cause an adverse effect on the timing considerations since the difference in processing time required to interface with a virtual bitstream and an actual USART is negligible. It is true that with an actual USART, the CPU might have to wait for a byte if the CPU were able to process bytes faster than the USART, or, if the CPU were slower than the USART, then a byte might be missed. However, this contingency was provided for by

conducting worst-case testing (see Section V.B for details). It is interesting to note that an analysis of test results showed that the 8080 CPU must execute an average of 574 instructions per received byte with a virtual bitstream and 573 instructions per received byte for an actual USART. Of these, 572 are identical, demonstrating the negligible difference between the two.

In addition to the virtual bitstream concept, a second aspect of the equipment test configuration had to be carefully thought out prior to programming and testing. This aspect was the matter of peripheral devices. The typical peripheral equipment configuration at a Mode I AUTODIN tributary usually consists of two magnetic tape drives (one for receive, one for transmit), a card reader, a card punch, a paper tape reader, and a line printer. Only two I/O ports, a teletype, and a CRT were available for testing. Since the teletype offered both a print capability for the receive function and a paper tape reader for the transmit function, it was selected over the CRT. The intention was to run tests of the algorithm using the teletype printer and reader simultaneously. This test was needed to check the correctness of the algorithm. However, the program was written so that, on incoming messages, the receiver would examine the SEL character, determine which output

device to select, select the output device, and then write the information on the selected output device (which in this case was always the teletype). In this way, the correctness of the algorithm could be tested without modifying the algorithm which would be used at an actual installation and without modifying the timing considerations. A similar argument holds for the transmit function: the program was made to check I/O ports for ready signals from nonexistent magnetic tape drives and card readers even though any actual input would always take place on the paper tape reader. Incorporating real hardware (such as magnetic tape drives) would require additional device driver routines and additional buffering. These requirements would increase the amount of main memory needed but would have a negligible impact on timing considerations.

By carefully considering all aspects of the hardware configuration prior to writing the program, it was possible to design a program which would be capable of being tested on the existing hardware but which also demonstrated the feasibility of a realistic AUTODIN tributary hardware configuration.

C. CHOOSING A PROGRAMMING LANGUAGE

PL/M, a block-structured, high-level systems language for the 8080 CPU was chosen as the language for developing the AUTODIN test program. There were four major reasons for choosing PL/M. The first reason was that the block structure and other logical constructs (such as if-then-else) facilitated the development of straightforward, efficient algorithms while freeing the programmer from unnecessary details which are often encountered in assembly language programming. The second reason was that, as a systems language, PL/M permits the programmer to control the 8080 just as closely as needed. Third, programs written in high-level languages are much easier to debug and maintain than large assembly language programs. Finally, the use of a high-level language would permit more rapid program development, an important consideration due to time constraints.

D. DESIGNING BY LEVELS

After defining the problem, developing an operational specification, understanding the hardware environment, and choosing a programming language, the next step that was taken was to begin designing the program in levels from the top down to the lowest levels. Much has been written

and spoken about structured programming and the top-down approach; however, in the author's opinion enough cannot be said. The author has used the top-down approach on several medium-size software projects with great success. Applying the approach to the AUTODIN test program also proved to be very successful: the entire project, from conception to successful testing took less than 15 weeks' part-time effort (see Section IV.G for details). It is believed that the reason for this success was due to using the top-down approach and modular, structured programming.

The highest level of the program was designed first, and the most time spent upon it. Correctness was insured at higher levels before proceeding to the design of lower ones. The reader may note that every procedure in the AUTODIN test program was labeled with a design level number. There were five design levels, with level one denoting the highest level and level five the lowest. As the design of the program began at the top level, it was discovered that the receive and transmit machines that were carefully developed in order to understand the AUTODIN protocol did not belong at the highest level of the program but rather at the second level. It became apparent that the actual implementation of these machines would require an operating system at the highest design level of the program to

coordinate and schedule the transmit and receive processes as well as other processes.

E. THE REQUISITE OPERATING SYSTEM

An analysis of the top level program requirements showed that, in addition to the transmit and receive processes, seven other processes were required to implement a functioning AUTODIN tributary. The nine processes are:

1. Receive logic process.
2. Transmit logic process (includes a reader process).
3. Poll peripheral devices process.
4. Poll receive side of USART process.
5. Poll transmit side of USART process.
6. Physical transmit process.
7. Writer process.
8. Operator input process.
9. Operator output process.

The functions of the receive and transmit processes were given in Chapter III of this thesis. The functions of the poll peripheral devices process were to poll the status of the local peripheral devices and to mark the devices as ready or not ready for input or output. Another important process was the poll receive side of USART process whose purpose was to indicate if a newly-received byte were in the USART

and ready for processing. Similarly, the poll transmit side of USART process had as its function to determine if the USART were ready to transmit the next byte. The purpose of the physical transmit process was to actually transfer bytes to the USART for transmission. The purpose of the writer process was to write incoming information onto the selected output device. The operator input process had as its function the input and interpretation of commands from the human operator. Finally, the operator output process had as its function the sending of alarm messages to the operator.

The management of these nine processes was the task of the highest level of the AUTODIN test program. It was necessary for this highest level to schedule the various processes and manage the corresponding peripheral and other devices. This scheduler in algorithmic form is shown below:

```
DO FOREVER:

    CALL POLL USART RECEIVER PROCESS;
    IF RECEIVE LOGIC PROCESS IS SCHEDULED OR
        RECEIVE LOGIC PROCESS DEVICE IS READY
        THEN CALL RECEIVE LOGIC PROCESS;
    IF WRITER PROCESS IS SCHEDULED AND WRITER
        PROCESS DEVICE IS READY
        THEN CALL WRITER PROCESS;
    IF OPERATOR INPUT PROCESS DEVICE IS READY
        THEN CALL OPERATOR INPUT PROCESS;
    IF OPERATOR OUTPUT PROCESS DEVICE IS READY
        AND OPERATOR OUTPUT PROCESS IS SCHEDULED
        THEN CALL OPERATOR OUTPUT PROCESS;
    CALL POLL USART TRANSMIT PROCESS;
```

```
    IF TRANSMIT LOGIC PROCESS DEVICE IS READY
        AND SENDING IS TRUE
        THEN CALL TRANSMIT LOGIC PROCESS;
    CALL POLL PERIPHERAL DEVICES;
END;
```

The above process scheduler was designed to utilize only polling to determine the status of devices. At first, some consideration was given to handling some of the devices (in particular, the receive side of the USART) on an interrupt basis. This could have been achieved since the 8080 CPU possesses an interrupt capability. However, careful analysis of the problem revealed that no advantage whatsoever was to be obtained from interrupt handling some or all of the devices. The main consideration was speed. When an incoming byte reaches the receive side of the USART, it remains there, ready for plucking by some process, for a time equal to eight times the reciprocal of the baud rate for synchronous operation and ten times the reciprocal of the baud rate for asynchronous operation. If the process scheduler can make one loop (performing all required tasks during this loop) and return to pluck the next byte from the receive side of the USART without ever losing a byte, then it will run fast enough to process a given baud rate. The rate at which the process scheduler can cycle through its DO FOREVER loop will be directly proportional to the baud rate it can handle, and this cycle rate is dependent upon the number of instructions

the CPU must perform per cycle. No gain in speed or efficiency can be obtained by interrupt processing in this case.

The actual implementation of the process scheduler may be found at the end of the AUTODIN program listing labeled program level one. It should be pointed out that the AUTODIN test program runs on the 8080 CPU without a resident operating system. In other words, the program contains its own, built-in operating system functions which consist of the process scheduler at level one of the program and the level five procedures which handle the actual input and output of the bytes. Program levels two, three, and four represent the various logic levels of the AUTODIN protocol and its associated processes such as writer, operator input, etc.

F. IMPLEMENTING THE RECEIVER AND TRANSMITTER PROCESSES

The next task to be performed in developing the AUTODIN test program was to implement the receiver and transmitter processes. These processes were well defined in Chapter III. Consequently, the task of implementing them was greatly simplified.

The receiver process (or RECEIVE$LOGIC, as it was called in the AUTODIN program) was designed to be a nine-state machine and was implemented as a level two procedure which

consisted of a nine-part case statement. Each invocation of the procedure corresponds to waking up of the receive logic process. Based on the input of a newly-received byte and the condition of various flags, the receive logic will perform designated actions and will make a state transition before going back to sleep.

The transmit process (or TRANSMIT$LOGIC, as it is called in the AUTODIN program) was designed to be a five-state machine and was implemented as a level two procedure which consisted of a five-part case statement. Each state (or case) was implemented as a level three procedure.

It is instructive to compare the state transition diagram of Figure 4 with the actual program as given in the listing. The procedure XMT$STATE$3 (contained in procedure TRANSMIT$LOGIC) corresponds to state three of Figure 4. One of two possible transitions will be made from state three. If the byte just read from the selected input device corresponds to a correct LMF character, then the transmit logic will place that character in the third text slot of the outgoing line block, perform a table lookup to find the corresponding SEL character, and place the SEL character in the second framing position of the outgoing line block. Then, the transmit logic will set its new state to four and go to sleep until reawakened by the process scheduler. On

the other hand, if the newly-read byte does not match with a correct LMF character then the transmit logic will send an alarm to the operator, cancel the current message, set its new state to one (the start state), and go to sleep.

The fact that the program was designed in levels is illustrated by pointing out that in this example the job scheduler and device manager are at level one, the transmit logic process is at level two, the actions of transmit state three are at level three, the procedure which checks LMF's for transmit state three is a level four, and the simple procedures which actually input and output bytes are at level five.

G. TESTING AND DEBUGGING THE PROGRAM

The testing and debugging of the AUTODIN test program was performed with relative ease, a fact the author attributes to the top-down approach. Of the 15 weeks spent on the project (from inception to successful testing), seven were spent defining the problem and designing the uppermost levels of the program, five were spent in coding and program development, and three were spent in testing and debugging the program on the Intellec 8. The definition of the problem and design of the upper levels have been discussed previously and consequently will not be discussed here.

Coding and program development were greatly facilitated by the use of an interactive, time-share terminal connected to the IBM 360 system of the Naval Postgraduate School. This terminal provided three invaluable tools for program development: a powerful context editor, a PL/M compiler, and an 8080 simulator (called Interp 80). These tools facilitated rapid program development and permitted the design-by-level approach by allowing testing of program modules at each level of development. Interp 80 was particularly useful in program development. For example, the AUTODIN receiver logic was tested to see if it could correctly recover from error conditions (such as incorrectly received line blocks). Using Interp 80, it was simple to introduce errors in order to test the performance of the receiver logic under various error conditions. Of the five bugs found during program development, four were found using Interp 80 before attempting to test on the Intellec 8. The five program errors discovered were contained in a program of over 1700 lines of source code. This translates into approximately one error per 350 lines of code -- proof that the top-down approach can produce good software.

In addition to the above problem, a timing problem was encountered during testing with the teletype. This was caused by the extremely slow reaction speed of the teletype

as compared to the 8080 CPU. The problem was rectified by inserting delays into the program. Upon completion, the object program was approximately 6100 bytes in size.

V. FEASIBILITY TESTING

As previously mentioned, the AUTODIN test program was designed to be tested in two ways. The first test was performed with actual peripheral devices to test the correctness of the algorithm, and the second test was performed with all devices virtual to obtain timing results on the 8080 CPU. Changing from one type of testing to the other was accomplished by changing one line of source code.

A. RESULTS OF THE PERIPHERAL DEVICE TEST

This program demonstrated its ability to simultaneously input and output information using the teletype printer and paper tape reader. In addition, the program demonstrates its ability to send alarm messages to the operator. Appendix B shows an actual test message which was sent on the Intellec 8.

B. RESULTS OF THE TIMING TESTS

Three timing tests were conducted. In each of them, 180,000 bytes were processed, and the time required for this to be done was recorded. This was actually accomplished by starting the program, using a stopwatch, and having the 8080 go into machine halt after 180,000 received bytes.

Appendix C shows the calculations used to obtain baud rates from these measurements.

The first timing test consisted of running the program with virtual peripheral devices for 180,000 bytes to obtain an average baud rate. During the test, message traffic was always being transmitted and received. Thus, during each cycle of level one, the program was required to receive one byte, write one byte, read one byte, and transmit one byte (in addition to polling all peripheral devices, even though they were not used). The result of this timing test was 3354 baud.

The second timing test was exactly the same as the first one with one difference. In order to make the AUTODIN test program capable of being run with both actual and virtual peripheral devices, it was necessary to make numerous checks throughout the program for the virtual or actual conditions. This required additional time. Consequently, these checks were removed, and the program was again timed. This time, the result was 3723 baud, a slightly faster rate, as expected.

The third timing test took into account worst case conditions as opposed to the average conditions of the first two tests. This test was necessary because the virtual USART was used. When using a virtual USART, no received byte is ever lost. Thus, even though an average baud rate of 3723

was measured, there might be worst case conditions where the receiver logic was going through its worst case (most time-consuming) processing coincident with the transmit logic doing the same, while at the same time, the operator input, operator output, and writer processes all required attention. Analysis of the AUTODIN test program revealed that, for the receiver logic, performing state nine actions (second control character of a two-character control sequence) were most time-consuming. For the transmit logic, state three actions (performing a LMF lookup and LMF-to-SEL conversion) were the most time-consuming. These actions were more time-consuming than error recovery. Under these conditions, an actual USART running at 3723 baud would result in lost received bytes. Therefore, it was necessary to conduct a worst-case test of the AUTODIN test program where these most time-consuming actions were repeatly performed. The result of this test was 2785 baud. An additional result was that, using an elapsed time of 517 seconds (See Appendix C) and an average instruction time of five microseconds, it was determined that the 8080 CPU executed an average of 574 instructions per received byte.

VI. CONCLUSIONS AND RECOMMENDATIONS

The AUTODIN test program is not an item of software ready for installation in AUTODIN tributary stations around the world. Rather, it was designed to demonstrate the feasibility of using a microcomputer to perform all of the functions required of a Mode I AUTODIN tributary station. In this regard, the AUTODIN test program fulfilled the purpose for which it was designed. By using conservative analysis techniques and taking into account worst-case processing requirements, it was shown that the 8080 CPU can perform all of the functions associated with an AUTODIN tributary station at modulation rates of 2400 baud. Furthermore, the AUTODIN test program required approximately 6000 (8-bit) words of main memory. Part of this memory requirement came from test parameters which need not be present in an actual working program. On the other hand, larger buffer sizes for interfacing with actual magnetic tape drives might be desirable. In addition, more main memory for certain nice-to-have features such as strings containing classification headings would be required. Nevertheless, it is conservatively estimated that 8194 words of main memory would handle the requirements for a fielded, working version of the program.

The result of all this is that AUTODIN communications can join the microcomputer revolution, and the revolution can be joined at a respectable baud rate of 2400. An 8080 CPU costs less than 30 dollars. An 8080 CPU, 8194 words of memory, power supply, cabinet, and I/O boards (with USART) cost less than one thousand dollars. There is absolutely no reason for continuing to lease expensive, large-scale computers at 60-80 thousand dollars per annum. It is true that the cost of peripherals must be added to the low cost of an 8080 based microcomputer system, but even with these costs added, the potential cost savings to the Federal Government are phenomenal. It is recommended that immediate attention be given to the official sanctioning and qualifying of microcomputers as DCS-approved equipment for AUTODIN use.

Another important implication of this thesis is the potential use of AUTODIN tributaries in mobile and tactical applications. Since microcomputers are so small and lightweight, they can be mounted in vehicles and aircraft to provide access to a worldwide digital information network. As defense management and weapon systems become more and more complex, the requirements for information in all forms (printed page, magnetic tape, floppy disk) at lower and lower echelons of command will increase. The use of microcomputers

will make it possible to expand the number of tributaries, giving more commanders at lower levels rapid access to the DCS. The field teletype can be replaced with a microcomputer connected to a lightweight line printer and (perhaps) a floppy disk unit, which will greatly improve the throughput rate and flexibility of communicated information. These are only a few of the potential applications. It is recommended that future development of field record communication systems take into account the use of microcomputers.

Finally, a side-product of this thesis was the state transition method of describing the AUTODIN protocol. This method proved to be vastly superior to the method used by DCA to describe AUTODIN. It is recommended that the state transition model be researched further, for it is felt that, with refinement, it could become a most effective method of describing communication protocols.

APPENDIX A

OPERATIONAL SPECIFICATION FOR AUTODIN TEST PROGRAM

I. SYSTEM OVERVIEW

The purpose of the test program shall be to investigate the feasibility of using a microcomputer such as the Intel 8080 (or equivalent) as a Mode I block-by-block AUTODIN tributary station. To demonstrate feasibility, it will be necessary to program the microcomputer to perform all of the functions that a tributary normally performs. These functions include the duplex, simultaneous transmission and reception of information via input from magnetic tape, card, or paper tape and output via line printer (or teletype), card, or magnetic tape. In the feasibility demonstration, the aforementioned peripheral devices may be real or virtual. In addition, the simultaneous transmission and reception of information over a full-duplex communication channel via a Universal Synchronous/Asynchronous Receiver/Transmitter (USART) must be accomplished. Furthermore, appropriate messages to the human operator must be sent whenever necessary. Although simultaneous transmission and reception is required, only one input device and one output device (which may be of the same or different type) may be selected

and in use at any point in time. In fact, the input device remains the same for each AUTODIN message transmitted; likewise for received messages and output devices. For example, the system might be simultaneously transmitting information from cards and receiving information which was being printed on a line printer.

In addition to performing the above tasks, the program must be designed such that the correctness of the algorithm may be tested by interfacing with actual peripheral devices. On the other hand, the program must be capable of being easily changed to work with virtual peripheral devices so that timing tests may be conducted. The reason for using virtual peripheral devices for timing tests is so that the central processing unit (CPU) of the microcomputer may run at full speed: the purpose of the program is to determine the speed at which a microcomputer can process AUTODIN messages and not to determine which input/output devices are rapid enough to function at AUTODIN tributaries.

III. PROCESSING REQUIREMENTS

A. RECEIVE PROCESSING

Incoming information arrives at the USART via a communication link and is transferred to the microprocessor, examined for parity correctness, stripped of control and

framing bytes, and transferred to the output device selected according to the SEL character in the incoming message. Acknowledgements (ACK1/ACK2) will be sent for correctly received line blocks; negative acknowledgements will be sent for incorrectly received line blocks. Synchronous idle will be recognizable by the receiver function, and notification of any loss of synchronization will be displayed to the operator.

B. TRANSMIT PROCESSING

When the human operator activates an input peripheral device for transmission by mounting a paper or magnetic tape or by loading a card deck into a card reader, the program must recognize that transmission is to begin. The program must begin transmission by reading the selected input device, building the line blocks for transmission, and must actually transmit the information, byte by byte, to the USART. Included in this operation is the insertion of proper parity and framing characters. In addition, the requisite coordination between the transmit and receive functions must occur so that proper channel coordination takes place according to the AUTODIN protocol.

APPENDIX B

ACTUAL TEST MESSAGE

```
RTTUZYUW RUWJAGC0000 2122200-UUUU--RUWJAGC.
ZNR UUUUU
R 302200 JUL 76
FM NTCC MONTEREY CA
TO NTCC MONTEREY CA
BT
UNCLAS//N00000//
THIS TEST MESSAGE DEMONSTRATES THE ABILITY OF THE 8080
CPU AND INTELLEC 8 MICROCOMPUTER TO PROCESS AUTODIN
MESSAGES.  ALTHOUGH THIS PARTICULAR MESSAGE IS PRINTED
ON A TELETYPE, THE 8080 IS CAPABLD OF INTERFACING WITH
OTHER PERIPHERAL DEVICES SUCH AS LINE PRINTERS, MAGNETIC
TAPE DRIVES, AND CARD PUNCHES/READERS.  THE POTENTIAL
USE OF MICROCOMPUTERS FOR AUTODIN APPLICATIONS HAS TWO
MAJOR IMPACTS:

     1)  CONSIDERABLE COST SAVINGS MAY RESULT FROM
         USING MICROCOMPUTERS IN PLACE OF LARGER,
         MORE EXPENSIVE COMPUTERS.

     2)  THE POSSIBLE USE OF LIGHTWEIGHT, RUGGEDIZED
         COMMUNICATIONS TRIBUTARIES FOR USE IN
         TACTICAL SITUATIONS CAN GREATLY IMPROVE
         FIELD RECORD COMMUNICATIONS.

END OF TEST MESSAGE.
BT
#0000




NNNN
```

APPENDIX C

TIMING TEST CALCULATIONS

TEST 1: CONSTANT CHECKING FOR VIRTUAL/ACTUAL DEVICES

ELAPSED TIME = 429.4 seconds

$$\frac{180,000 \text{ bytes}}{429.4 \text{ seconds}} \times 8 \text{ bits/byte} = 3354(\pm 25) \text{ baud}$$

TEST 2: NO CHECKING FOR VIRTUAL/ACTUAL DEVICES

ELAPSED TIME = 386.8 seconds

$$\frac{180,000 \text{ bytes}}{386.8 \text{ seconds}} \times 8 \text{ bits/byte} = 3723(\pm 25) \text{ baud}$$

TEST 3: WORST CASE PROCESSING

ELAPSED TIME = 517.0 seconds

$$\frac{180,000 \text{ bytes}}{517.0 \text{ seconds}} \times 8 \text{ bits/byte} = 2785(\pm 25) \text{ baud}$$

```
100H:     /*  THE PROGRAM WILL BE LOADED IN THE 8080
               BEGINNING AT 100 HEX        */


    /*                    A U T O D I N

PROGRAMMER: GORDON E. ANDERSON.
SOURCE CCMPUTER: IBM 360 (USED FOR COMPILATION).
OBJECT CCMPUTER: INTELLEC 8 (USED FOR EXECUTION).
CLASSIFICATION: UNCLASSIFIED.
SOURCE LANGUAGE: PLM.
DATES: AUGUST - NOVEMBER 1976.

    THE PURPOSE OF THIS PROGRAM IS TO DEMONSTRATE THE FEASABILITY
OF USING A MICROCOMPUTER, SUCH AS THE INTELLEC 8 WITH 8080 CPU, FOR
A DEFENSE COMMUNICATION SYSTEM (DCS) MODE I AUTODIN TRIBUTARY STATION
(BLOCK EY BLOCK MODE OF OPERATION).  THE PROGRAM IS DESIGNED TO PERFORM
ALL OPERATOR INTERFACES, LOCAL PERIPHERAL INTERFACES, COMMUNICATION
INTERFACES, AS WELL AS THE LOGICAL PROTOCOL INTERFACE WITH AN
AUTOMATIC SWITCHING CENTER (ASC).  THE PROGRAM WILL PERFORM THIS
INTERFACING INDEPENDENT OF THE RATE OF INFORMATION TRANSFER
(BAUD RATE OR BITS PER SECOND).  THIS WILL PERMIT TESTING
OF THE PROGRAM AT DIFFERENT BAUD RATES IN ORDER TO DETERMINE
THE MAXIMUM SPEED AT WHICH THE MICROCOMPUTER CAN PROCESS
AUTODIN TRAFFIC.

    THIS PROGRAM RUNS ON THE "BARE MACHINE", THAT IS,
IT DOES NOT REQUIRE A RESIDENT OPERATING SYSTEM; RATHER,
IT CONTAINS ITS OWN OPERATING SYSTEM WHICH PERMITS MULTI-
PROCESSING OF THE INDEPENDENT PROCESSES WHICH MAKE UP THE
TOTAL AUTODIN PROGRAM.  PROCESS SCHEDULING AND DEVICE MANAGE-
MENT ARE PERFORMED AT THE HIGHEST LEVEL OF THE PROGRAM.
THE NINE MAJOR PROCESSES WHICH ARE SCHEDULED ARE:

    1.   POLL$USAR
    2.   RECEIVE$LOGIC
    3.   WRITER
    4.   OPERATOR$INPUT
    5.   OPERATOR$OUTPUT
    6.   TRANSMIT$LOGIC
    7.   TRANSMITTER
    8.   POLL$DEVICES
    9.   POLL$USAT
```

```
    IT IS IMPORTANT TO NOTE THAT EACH PROCESS IS PERMIT-
TED TO RUN FOR ONE BYTE AT A TIME; CONSEQUENTLY, SIMULTANEOUS
TRANSMISSION, RECEPTION, INPUT, AND OUTPUT MAY TAKE PLACE.

    WHEN READING THE PROGRAM, BEGIN AT THE HIGHEST LEVEL,
LEVEL 1, AND WORK THROUGH THE LEVELS 1, 2, 3, ...ETC.,
SINCE THE HIGHER LEVEL PROCEDURES INVOKE THOSE OF LOWER
LEVEL (AND LARGER NUMBER). ALL PROCEDURES ARE LABELLED
WITH THEIR LEVEL NUMBER.  THE HIGHEST LEVEL, LEVEL 1,
IS LOCATED AT THE BOTTOM OF THE PROGRAM LISTING.


    FINALLY, EXTENSIVE USE OF THE LITERAL SUBSTITUTION CAPABILITY
OF PLM HAS BEEN MADE.  THIS PERMITS THE PROGRAM TO BE READ IN
ALGORITHMIC ENGLISH FORM BY THOSE WHO ARE INTERESTED IN
ITS LOGICAL STRUCTURE.   THOSE INTERESTED IN BIT MANIPULATIONS
AND OTHER DETAILS MAY STUDY THE LITERAL DECLARATIONS.     */



    /*   **********   LITERAL DECLARATIONS   **********   */

DECLARE
    /* THE FOLLOWING LITERALS ARE USED FOR READABILITY */

    TRUE LITERALLY '1',
    FALSE LITERALLY '0',
    SPACE LITERALLY '20H',
    FOREVER LITERALLY 'WHILE TRUE',
    NOT$READY LITERALLY '0',
    READY LITERALLY '1',
    BOOT LITERALLY '0',
    SCHEDULED LITERALLY '0',
    DEVICE$READY LITERALLY '1',
    LF LITERALLY '10',
    CR LITERALLY '13',
    EXPIRED LITERALLY '> 600',
    EVEN LITERALLY ' ',
    LOW$7$MASK LITERALLY '7FH',    /* USED TO REMOVE PARITY BITS    */
    PARITY$MASK LITERALLY '80H',   /* USED TO INSERT PARITY BITS    */
    TTY LITERALLY '0',             /* USED FOR TELETYPE PORT        */
    MAG$TAPE LITERALLY '2',        /* USED FOR MAG TAPE I/O PORT    */
    CARD$PUNCH LITERALLY '3',      /* USED FOR CARD PUNCH I/O PORT  */
    TTY$INFO LITERALLY '0',        /* USED FOR TTY I/O PORT         */
    TTY$STATUS LITERALLY '1',      /* USED FOR TTY STATUS PORT      */
    CRT$INFO LITERALLY '0',        /* USED FOR CRT I/O PORT         */
    CRT$STATUS LITERALLY '1',      /* USED FOR CRT STATUS PORT      */
```

```
    USART$INFO LITERALLY '4',          /* USED FOR USART I/O PORT        */
    USART$STATUS LITERALLY '5',        /* USED FOR USART STATUS PORT     */
    NOT$SELECTED LITERALLY '255',      /* DENOTES NO DEVICE SELECTED     */
    BELL LITERALLY '07H',              /* ALERTS THE HUMAN OPERATOR      */

    /* THE FOLLOWING LITERAL DECLARATIONS REPRESENT THE CONTROL
       AND FRAMING CHARACTERS USED IN AUTODIN   */

    SOH LITERALLY '81H',          /* START OF HEADER LINE  BLOCK          */
    STX LITERALLY '82H',          /* START OF INTERMEDIATE LINE BLOCK     */
    ETX LITERALLY '03H',          /* END OF LAST LINE BLOCK               */
    ACK1 LITERALLY '06H',         /* ACKNOWLEDGEMENT ONE                  */
    INV LITERALLY '87H',          /* INVALID ACK OR NAK RECEIVED          */
    REP LITERALLY '11H',          /* REQUEST FOR ANSWER                   */
    RM LITERALLY '12H',           /* REJECT YOUR MESSAGE                  */
    NAK LITERALLY '95H',          /* NEGATIVE ACKNOWLEDGEMENT             */
    SYN LITERALLY '96H',          /* SYNCHRONOUS IDLE PATTERN             */
    ETB LITERALLY '17H',          /* END OF INTERMEDIATE LINE BLOCK       */
    CAN LITERALLY '18H',          /* CANCEL THIS MESSAGE                  */
    EM LITERALLY '99H',           /* END OF TEXT MARKER                   */
    MC LITERALLY '9AH',           /* MODE CHANGE                          */
    ACK2 LITERALLY '9CH',         /* ACKNOWLEDGEMENT TWO                  */
    WBT LITERALLY '1EH',          /* WAIT REQUEST                         */
    DEL LITERALLY '0FFH';         /* DELETED FRAMING CHARACTER            */


    /*    **********   GLOBAL VARIABLE DECLARATIONS   **********   */


DECLARE

    /* THE FOLLOWING VECTORS ARE USED BY THE EXECUTIVE
       TO SCHEDULE THE DESCRIBED PROCESSES    */

    RECEIVE$LOGIC$PROCESS (2) BYTE,
    WRITER$PROCESS (2) BYTE,
    OPERATOR$INPUT$PROCESS (2) BYTE,
    OPERATOR$OUTPUT$PROCESS (2) BYTE,
    TRANSMIT$LOGIC$PROCESS (2) BYTE,
    TRANSMITTER$PROCESS (2) BYTE;


    /* GLOBAL FLAG AND VARIABLE DECLARATIONS */

DECLARE

    USAR$CHECK BYTE,     /* USED TO DETECT IF THE USAR HAS GONE DEAD  */
    RCV$BYTE BYTE,       /* THE BYTE JUST RECEIVED FROM THE USAR      */
```

```
RCV$STATE BYTE,        /* THE STATE OF THE RECEIVER LOGIC PROCESS    */
RCV$BP BYTE,           /* THE BLOCK PARITY OF THE  ECEIVED BLOCK     */
SYN$CTR BYTE,          /* COUNTS THE NUMBER OF SYN RECEIVED IN A ROW*/
PREV$RCV$BYTE BYTE,    /* SAVES THE PREVIOUSLY RECEIVED BYTE         */
MC$FLAG BYTE,          /* TRUE=DATA FORMAT MODE CHANGE; FALSE O.W.   */
WBT$CTR BYTE,          /* COUNTS NUMBER OF WBT RECEIVED IN A ROW     */
NAK$CTR BYTE,          /* COUNTS NUMBER OF NAK RECEIVED IN A ROW     */
CHECK$RCV$TIMER BYTE, /* TRUE MEANS CHECK THE RECEIVE TIMER          */
RCV$SYNCH$TIMER ADDRESS, /*IF TIMER GOES OFF, SYNCH HAS BEEN LOST*/
RCV$MID$MSG BYTE,      /* TRUE WHEN RCVR IS IN THE MIDDLE OF A MSG   */
RCV$TXT$CTR BYTE,      /* COUNTS THE TEXT FOR INCOMING BLOCKS        */
RCV$ERROR BYTE,        /* TRUE IF AN ERROR HAS BEEN FOUND IN BLOCK   */
PREV$RCV$STATE BYTE,/* SAVES THE PREVIOUS STATE OF THE RCVR          */
RCV$ACK$1$2 BYTE,      /* 1=ACK1 TO BE RCVD NEXT; 2=ACK2 NEXT        */
BLOCK$START$STATE BYTE, /* RCVR STATE AT BEGINNING OF BLOCK          */
RCV$REP$CTR BYTE,      /*  COUNTS NUMBER OF REP RECEIVED IN A ROW    */
OUTPUT$DEVICE BYTE, /* RECEIVER SELECTS BASED ON RCVD SEL            */
STILL$WRITING BYTE, /* TRUE MEANS WRITER IS STILL WRITING            */

XMT$STATE BYTE,        /* THE STATE OF THE TRANSMIT LOGIC PROCESS    */
XMT$BYTE BYTE,         /* THE BYTE JUST READ IN AND TO BE XMTTED     */
XMT$BP BYTE,           /* KEEPS TRACK OF BLOCK PARITY FOR XMTR       */
XMT$TXT$CTR BYTE,      /* COUNTS TEXT FOR OUTGOING LINE BLOCKS       */
XMT$WAIT BYTE,         /* TRUE= XMTR WAITING FOR ACK'S               */
XMT$ACK$1$2 BYTE,      /* 1= SEND ACK1 NEXT, 2= SEND ACK2 NEXT       */
XMT$ANS$TIMER ADDRESS, /* IF EXPIRED, ANSWER NOT RCVD SOON ENOUGH*/
CHECK$ANS$TIMER BYTE,     /* TRUE MEANS TO CHECK THE XMT TIMER       */
XMT$REP$CTR  BYTE,    /* KEEPS TRACK OF  REPS SENT IN A ROW          */
INPUT$DEVICE BYTE,   /* READ THIS DEVICE; TRANSMIT IT'S DATA         */
SENDING BYTE,          /* TRUE MEANS WE'RE SENDING A MSG NOW         */
CAN$FLAG BYTE,         /* TRUE MEANS WE'RE CANCELLING XMTD MSG       */
AWAITING$ACK BYTE,   /* TRUE=WAITING FOR LAST BLOCK TO BE ACKD       */

OUTPUT$STATE BYTE,   /* USED TO PERFORM OPR OUTPUTS                  */
ALARM$MSG BYTE,      /* USED TO SEND ONE LTR AIARM TO OPR            */
EOM$STATE BYTE,      /* USED TO CHECK FOR END OF MSG 'EOM'           */
LF$CTR BYTE,         /* USED TO COUNT LINE FEEDS (LOOKS FOR 8)       */
N$CTR BYTE,          /* USED TO COUNT N'S (LOOKS FOR 4 IN A ROW)     */
VIRTUAL BYTE,        /* TRUE MEANS READ/WRITE IN CORE; FALSE MEANS
                        READ/WRITE ON ACTUAL PERIPHERAL DEVICES      */

RW$BUFFER$1 (81) BYTE,   /* UTILITY BUFFER FOR RECEIVE LOGIC AND
                            WRITER PROCESS                           */
RW$BUFFER$2 (81) BYTE,   /* SAME PURPOSE AS RW$BUFFER$1              */
RCV$BUFFER$ADDR ADDRESS, /* SETS ADDR OF THE CURRENT RCV BUFFER      */
(RCV$BUFFER BASED RCV$BUFFER$ADDR) (81) BYTE,
                        /* CURRENT BUFFER USED BY RCV LOGIC          */
WRITE$BUFFER$ADDR ADDRESS,
```

```
                              /* ADDRESS OF CURRENT WRITE BUFFER       */
    (WRITE$BUFFER BASED WRITE$BUFFER$ADDR) (81) BYTE,
                              /* CURRENT BUFFER FOR WRITE PROCESS      */
    W$BUFFER$PTR BYTE,        /* POINTER FOR WRITE$BUFFER              */

    T$BUFFER$1 (85) BYTE,     /* TWO UTILITY BUFFERS USED FOR XMT      */
    T$BUFFER$2 (85) BYTE,     /* AND TRANSMITTER PROCESSES             */
    XMT$LOGIC$BUFFER$ADDR ADDRESS,
                              /* SETS ADDR OF CURRENT XMT LOGIC BUFFER*/
    (XMT$LOGIC$BUFFER BASED XMT$LOGIC$BUFFER$ADDR) (85) BYTE,
                              /* CURRENT BUFFER USED BY XMT LOGIC      */
    TRANSMIT$BUFFER$ADDR ADDRESS,
                              /* SETS ADDR OF CURRENT TRANSMIT BUFFER */
    (TRANSMIT$BUFFER BASED TRANSMIT$BUFFER$ADDR) (85) BYTE,
                              /* THE CURRENT TRANSMIT PROCESS BUFFER  */
    T$BUFFER$PTR BYTE,        /* POINTER FOR TRANSMIT BUFFER           */
    T$BUFFER$LENGTH BYTE,     /* CURRENT LENGTH OF TRANSMIT BUFFER     */

    CONCHAR$BUFFER (8) BYTE,  /* BUFFER FOR THE CONTROL CHARS TO SEND */
    CC$PTR1 BYTE,             /* POINTER FOR CONTROL CHAR BUFFER       */
    CC$PTR2 BYTE,             /* POINTER FOR CONTROL CHAR BUFFER       */

    /* THE FOLLOWING FOUR VECTORS ARE USED TO STORE RECEIVE
       LOGIC TABLE LOOKUP ACTIONS*/

    TABLE$1 DATA(0,1,2,2,2,2,3,3,3,3,2,2,3,4,2,3,2,2,2,3,2,3,2),
    TABLE$2 DATA(0,2,5,2,2,2,3,3,3,3,2,2,3,4,2,3,2,2,2,3,2,3,2),
    TABLE$3 DATA(0,6,6,6,6,6,3,3,3,3,6,6,3,6,6,3,7,8,6,3,6,3,9),
    TABLE$4 DATA(0,10,10,11,10,10,3,3,3,3,10,10,3,12,13,3,10,10,
                 10,3,10,3,10),

    /* THE FOLLOWING TWO VECTORS ARE USED TO STORE THE CORRECT
        'SEL' CHARACTERS SO THAT THE TRANSMIT LOGIC CAN INSERT
        THE CORRECT 'SEL' CHARACTER BASED ON THE LMF CHARACTER
        OF THE MESSAGE BEING TRANSMITTED  */

    SEL$LOOKUP$1 DATA('A','B','D','C',0,'H','A',0,'B'),
    SEL$LOOKUP$2 DATA('A','A','C','H');


    /* DECLARATIONS FOR TEST AND DEBUG */

DECLARE

     /* THE FOLLOWING DECLARATIONS ARE FOR ODD PARITY TEXT
        CHARACTERS WHICH ARE USED IN A TEST MESSAGE          */
```

```
OPA LITERALLY '0C1H',
OPB LITERALLY '0C2H',
OPC LITERALLY '043H',
OPD LITERALLY '0C4H',
OPE LITERALLY '045H',
OPF LITERALLY '046H',
OPG LITERALLY '0C7H',
OPH LITERALLY '0C8H',
OPI LITERALLY '049H',
OPJ LITERALLY '04AH',
OPK LITERALLY '0CBH',
OPL LITERALLY '04CH',
OPM LITERALLY '0CDH',
OPN LITERALLY '0CEH',
OPO LITERALLY '04FH',
OPP LITERALLY '0D0H',
OPQ LITERALLY '051H',
OPR LITERALLY '052H',
OPS LITERALLY '0D3H',
OPT LITERALLY '054H',
OPU LITERALLY '0D5H',
OPV LITERALLY '0D6H',
OPW LITERALLY '057H',
OPX LITERALLY '058H',
OPY LITERALLY '0D9H',
OPZ LITERALLY '0DAH',
OP0 LITERALLY '0B0H',
OP1 LITERALLY '031H',
OP2 LITERALLY '032H',
OP3 LITERALLY '0B3H',
OP4 LITERALLY '034H',
OP5 LITERALLY '0B5H',
OP6 LITERALLY '0B6H',
OP7 LITERALLY '037H',
OP8 LITERALLY '038H',
OP9 LITERALLY '0B9H',
OPDA LITERALLY '0ADH',
OPCR LITERALLY '00DH',
OPLF LITERALLY '08AH',
OPSL LITERALLY '02FH',
OPPD LITERALLY '023H',
OPPE LITERALLY '0AEH',
OPSE LITERALLY '020H',
EPA LITERALLY '041H';

/* TEST$MSG IS USED WHEN A TEST MESSAGE IS DESIRED FOR
```

```
        TESTING THE RECEIVER AUTOMATON BY ITSELF                  */      AUT02890
                                                                          AUT02900
DECLARE TESTSMSG DATA(SYN,                                                AUT02910
SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN,                         AUT02920
SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN, SOH, EPA,                         AUT02930
OPR, OPI, OPT, OPU, OPZ, OPY, OPU, OPW, OPSP,OPR,                         AUT02940
OPU, OPW, OPJ, OPA, OPG, OPC, OPO, OPO, OPO, OPO,                         AUT02950
OPSP,OP2, OP1, OP2, OP2, OP2, OPO, OPO, OPDA,OPU,                         AUT02960
OPU, OPU, OPU, OPDA,OPDA,OPR, OPU, OPW, OPJ, OPA,                         AUT02970
OPG, OPC, OPPE,OPCR,OPLF,OPZ, OPN, OPR, OPSP,OPU,                         AUT02980
OPU, OPU, OPU, OPU, OPCR,OPLF,OPR, OPSP,OP3, OPO,                         AUT02990
OP2, OP2, OP2, OPO, OPO, OPZ, OPSP,OPJ, OPU, OPL,                         AUT03000
OPSP,OP7, OP6, OPCR,OPLF,OPF, OPM, OPSP,OPN,OPT,                          AUT03010
ETB,   0, SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN,                         AUT03020
SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN, STX, DEI,                         AUT03030
OPC, OPC, OPSP,OPM, OPO, OPN, OPT, OPE, OPR, OPE,                         AUT03040
OPY, OPCR,OPLF,OPT, OPO, OPSP,OPN, OPT, OPC, OPC,                         AUT03050
OPSP,OPF, OPO, OPN, OPT, OPE, OPR, OPE, OPY, OPCR,                        AUT03060
OPLF,OPE, OPT, OPCR,OPLF,OPU, OPN, OPC, OPL, OPA,                         AUT03070
OPS, OPSL,OPSL,OPN, OPO, OPO, OPO, OPO, OPO, OPSP,                        AUT03080
OPSL,OPCR,OPLF,OPN, OPO, OPW, OPSP,OPI, OPS, OPSP,                        AUT03090
OPT, OPH, OPE, OPSP,OPT, OPI, OPM, OPE, OPSP,OPF,                         AUT03100
OPO, OPR, OPSP,OPA, OPL, OPL, OPSP,OPG, OPO, OPO,                         AUT03110
ETB,   0, SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN,                         AUT03120
SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN, STX, DEI,                         AUT03130
OPD, OPSP,OPM, OPE, OPN, OPSP,OPT, OPO, OPSP,OPC,                         AUT03140
OPO, OPE, OPE, OPSP,OPT, OPO, OPSP, OPT, OPH,OPE,                         AUT03150
OPSP,OPA, OPI, OPD, OPSP,OPO, OPF, OPT, OPH, OPE,                         AUT03160
OPI, OPR, OPSP,OPC, OPO, OPU, OPN, OPT, OPR, OPY,                         AUT03170
OPPE,OPCR,OPLF,OPB, OPT, OPCR,OPLF,OPPD,OPO, OPO,                         AUT03180
OPO, OPO, OPCR,OPLF,OPLF,OPLF,OPLF,OPLF,OPLF,OPLF,                        AUT03190
OPLF,OPN, OPN, OPN, OPN,  EM, ETX,   0, SYN, SYN,                         AUT03200
SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN, SYN);                        AUT03210
                                                                          AUT03220
    /* TESTSTAPE IS USED FOR TESTING THE TRANSMITTER - IT                 AUT03230
       PROVIDES A VIRTUAL PAPER TAPE FOR THE TRANSMIT LOGIC */            AUT03240
                                                                          AUT03250
DECLARE TESTSTAPE DATA(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',  AUT03260
'R','T','T','U','Z','Y','U','W',' ','R',                                  AUT03270
'R','U','W','J','A','G','C','O','O','O',                                  AUT03280
'O','2','1','2','2','2','O','O','-','U',                                  AUT03290
'U','U','U','-','-','R','U','W','J','A',                                  AUT03300
'G','C',ODH,OAH,'A','U','T','O','D','I',                                  AUT03310
'N',' ','T','E','S','T',' ','M','E','S',                                  AUT03320
'S','A','G','E',' ','A','B','C','D','E',                                  AUT03330
'F','G','H','I','J','K','L','M','N','O',                                  AUT03340
'P','Q','R','S','T','U','V','W','X','Y',                                  AUT03350
'Z','0','1','2',OAH,OAH,OAH,OAH,OAH,OAH,                                  AUT03360
```

```
OAH,OAH,'N','N','N','N');
DECLARE R$TEST$PTR ADDRESS;

    /* PAPER IS A VECTOR WHICH SERVES AS A VIRTUAL PAPER */

DECLARE PAPER(401) BYTE;

    /* THE FOLLOWING DECLARATIONS ARE FOR POINTERS WHICH
       REQUIRED FOR THE VIRTUAL BUFFERS PREVIOUSLY DECLARED */

DECLARE S$TEST$PTR ADDRESS;
DECLARE T$TEST$PTR BYTE;
DECLARE PAPER$PTR ADDRESS;
DECLARE BYTE$COUNTER ADDRESS;
DECLARE TEST$COUNTER BYTE;
DECLARE DEBUG$FLAG BYTE;


    /* END OF TEST/DEBUG DECLARATIONS */



    /* ********************************************************* */
    /* *****      GLOBAL UTILITY PROCEDURES      *************** */

CRT$OUT:PROCEDURE(CHARACTER);
    /* LEVEL 5 PROCEDURE */

    /* SENDS ONE CHARACTER TO THE CONSOLE DEVICE */

     DECLARE CHARACTER BYTE;

    IF NOT VIRTUAL THEN
        DO;
            DO WHILE ROR(INPUT(CRT$STATUS),2);
            END;  /* WAIT UNTIL CONSOLE IS READY */
            OUTPUT(CRT$INFO)=NOT CHARACTER;
        END;

END CRT$OUT;

CRT$IN: PROCEDURE BYTE;

    /* LEVEL 5 PROCEDURE */

    /* INPUTS ONE BYTE FROM THE CONSOLE DEVICE */

    RETURN (NOT INPUT(CRT$INFO)) AND LOW$7$MASK;
```

```
END CRT$IN;

SEND$CC: PROCEDURE(CONTROL$CHAR);

    /* LEVEL 5 PROCEDURE */

    /* EUTS TWO CONTROL CHARACTERS IN THE BUFFER  TO BE SENT */

    IECLARE CONTROL$CHAR BYTE;

    CONCHAR$BUFFER(CC$PTR2),CONCHAR$BUFFER(CC$PTR2 + 1)=CONTROL$CHAR;
    IF (CC$PTR2:=CC$PTR2 + 2) > 7 THEN
        CC$PTR2=0;

END SENI$CC;

INITIALIZE$RCVR: PROCEDURE;

    /* LEVEL 5 PROCEDURE */

    /* INITIALIZES THE RECEIVE LOGIC AND WRITER PROCESSES */

    CHECK$RCV$TIMER=TRUE;
    MC$FLAG, RCV$MID$MSG, RCV$ERROR, STILL$WRITING=FALSE;

    XMT$ACK$1$2, W$BUFFER$PTR, PREV$RCV$STATE, BLOCK$START$STATE=1;

    RCV$BP, SYN$CTR, WBT$CTR, NAK$CTR, RCV$SYNCH$TIMER,
    RCV$TXT$CTR, RCV$REP$CTR, RW$BUFFER$1, RW$BUFFER$2=0;

    CUTEUT$DEVICE=NOT$SELECTED;

    RCV$EUFFER$ADDR,WRITE$BUFFER$ADDR=.RW$BUFFER$1;

    RCV$STATE=5;

    RECEIVE$LOGIC$PROCESS(SCHEDULED)=TRUE;

    WRITER$PROCESS(SCHEDULED)=FALSE;

    IF NOT VIRTUAL THEN
        DO;
            RCV$STATE=1;
            RECEIVE$LOGIC$PROCESS(SCHEDULED)=FAISE;
        END;

END INITIALIZE$RCVR;
```

```
INITIALIZE$XMTR: PROCEDURE;

    /* LEVEL 5 PROCEDURE */

    /* INITIALIZES THE TRANSMIT LOGIC AND TRANSMITTER PROCESSES */

    XMT$WAIT, CHECK$ANS$TIMER, CAN$FLAG, SENDING,
    AWAITING$ACK=FALSE;

    RCV$ACK$1$2, XMT$STATE, T$BUFFER$PTR, EOM$STATE=1;

    XMT$BP, XMT$ANS$TIMER, XMT$REP$CTR, T$BUFFER$1,
    T$BUFFER$2, LF$CTR, N$CTR=0;

    INPUT$DEVICE=NOT$SELECTED;

    XMT$LOGIC$BUFFER$ADDR=.T$BUFFER$2;

    TRANSMIT$BUFFER$ADDR=.T$BUFFER$1;

    TRANSMIT$LOGIC$PROCESS(DEVICE$READY)=FALSE;

END INITIALIZE$XMTR;

ALARM: PROCEDURE (CONDITION);

     /* LEVEL 5 PROCEDURE */

    /* SENDS AN APPROPRIATE ALARM MESSAGE BY SCHEDULING THE
    OPERATOR OUTPUT PROCESS. THE ALARM CONDITION IS COMMUNICATED
    VIA GLOBAL VARIABLE 'ALARM$MSG'.  RE-INITIALIZATION OF
    OF THE RECEIVER OR TRANSMITTER IS PERFORMED WHEN NECESSARY. */

    DECLARE CONDITION BYTE;
    DO CASE CONDITION;

        ; /* CASE 0 NOT USED */

        DO;  /* CASE 1 -- 3 NAK'S RECEIVED IN A ROW */
             CALL SEND$CC(CAN);
             ALARM$MSG='N';
             CALL INITIALIZE$XMTR;
        END;

        DO;  /* CASE 2 - - 3 WBT'S RCVD IN A ROW */
             CALL SEND$CC(CAN);
             ALARM$MSG='W';
```

```
            CALL INITIALIZE$XMTR;
        END;

        DO;   /* CASE 3 -- CAN RCVD FROM THE ASC */
            ALARM$MSG='C';
            CALL INITIALIZE$RCVR;
        END;

              /* CASE 4 -- REP RECEIVED BETWEEN MSGS */
        ALARM$MSG='R';

        DO;   /* CASE 5 -- RM RECEIVED */
            CALL SEND$CC(CAN);
            ALARM$MSG='M';
            CALL INITIALIZE$RCVR;
        END;

              /* CASE 6 -- INV RCVD */
        ALARM$MSG='I';

        DO;   /* CASE 7 -- 3 REP'S SENT IN A ROW */
            CALL SEND$CC(CAN);
            ALARM$MSG='3';
            CALL INITIALIZE$XMTR;
        END;

        DO;   /* CASE 8 -- INVALID LMF ON OUTGOING MSG */
            ALARM$MSG='L';
            CALL INITIALIZE$XMTR;
        END;

              /* CASE 9 FLASH; SEND BELL AND F TO CONSOLE */
        ALARM$MSG='F';

              /* CASE 10 - LOSS OF SYNCH AND SYNC TIMER EXPD */
        ALARM$MSG='S';

              /* CASE 11 - USAR DEAD */
        ALARM$MSG='D';

    END; /* OF CASE CONDITION */

    OUTPUT$STATE=1;
    OPERATOR$OUTPUT$PROCESS(SCHEDULED)=TRUE;

END ALARM;

GET$RCVD$BYTE: PROCEDURE BYTE;
```

```
        /* LEVEL 5 PROCEDURE */

        RCV$SYNCH$TIMER=RCV$SYNCH$TIMER + 1;
        IF CHECK$ANS$TIMER THEN
             XMT$ANS$TIMER=XMT$ANS$TIMER + 1;
        ELSE XMT$ANS$TIMER=0;
        /* THESE TWO COUNTERS ARE BUMPED EACH TIME A BYTE IS RECEIVED.
        WHEN THEY GET OVER 600 IN VALUE, THE TIMERS HAVE EXPIRED.  */
        USAR$CHECK=0;   /* JUST GOT A BYTE; SO USAR IS GOOD   */


IF VIRTUAL THEN
     DO;
     BYTE$COUNTER=BYTE$COUNTER + 1;
     IF(R$TEST$PTR:=R$TEST$PTR + 1) > 300 THEN
          R$TEST$PTR=0;
     RETURN TEST$MSG(R$TEST$PTR);

     END;
ELSE RETURN NOT(INPUT(USART$INFO));

END GET$RCVD$BYTE;

WRITE$BYTE: PROCEDURE (CHARACTER);

        /* LEVEL 5 PROCEDURE */

        DECLARE CHARACTER, BYTE;

IF VIRTUAL THEN
     DO;
     PAPER(PAPER$PTR)=CHARACTER;
     PAPER$PTR=PAPER$PTR + 1;
     IF PAPER$PTR > 400 THEN
          PAPER$PTR=0;

     END;
ELSE
     DO;
          IF OUTPUT$DEVICE=TTY THEN
               DO;
                    DO WHILE ROR(INPUT(TTY$STATUS),2);
                    END;     /* WAIT UNTIL TTY IS READY */
                    OUTPUT(TTY)=NOT CHARACTER;
               END;
          ELSE
               IF OUTPUT$DEVICE=MAG$TAPE THEN
```

```
                    OUTPUT(MAG$TAPE)=NOT CHARACTER;
                ELSE OUTPUT(CARD$PUNCH)=NOT CHARACTER;
    END;

END WRITE$BYTE;

SEND$BYTE: PROCEDURE (CHARACTER);

     /* LEVEL 5 PROCEDURE */

     DECLARE CHARACTER BYTE;


IF VIRTUAL THEN
    DO;

    IF(S$TEST$PTR:=S$TEST$PTR + 1)>300 THEN
         S$TEST$PTR=0;
    TEST$MSG(S$TEST$PTR)=CHARACTER;
    END;
ELSE
    OUTPUT(USART$INFO)=NOT CHARACTER;

END SEND$BYTE;


GET$BYTE$TO$XMT: PROCEDURE BYTE;

     /* LEVEL 5 PROCEDURE */

IF VIRTUAL THEN
    DO;
    IF (T$TEST$PTR:=T$TEST$PTR + 1) > 115 THEN
          T$TEST$PTR=0;
    RETURN TEST$TAPE(T$TEST$PTR);
    END;
ELSE
    DO;
         IF INPUT$DEVICE=TTY THEN
              DO;
                   IF INPUT(TTY$STATUS) THEN /* STROBE READER */
                        DO;
                             OUTPUT(TTY$STATUS)=1;
                             OUTPUT(TTY$STATUS)=0;
                        END;
                   DO WHILE INPUT(TTY$STATUS);
                   END;  /* WAIT UNTIL TTY IS READY */
                   RETURN(NOT INPUT (TTY$INFO) AND LOW$7$MASK);
```

```
            END;
        ELSE
            IF INPUT$DEVICE=MAG$TAPE THEN
                RETURN (NOT(INPUT(MAG$TAPE)) AND LOW$7$MASK);
            ELSE
                RETURN (NOT(INPUT(CARD$PUNCH)) AND LOW$7$MASK);
    END;

END GET$BYTE$TO$XMT;

    /*     ****************************************************   */
    /*      END OF THE LEVEL 5 UTILITY PROCEDURES                 */

    /*     ****************************************************   */
    /*    THE 10 MAJOR PROCESSES (LEVEL 2 PROCEDURES) ARE LISTED BELOW*/

INITIALIZE: PROCEDURE;
    /* LEVEL 2 PROCEDURE */

    /* INITIALIZES EVERYTHING AT POWER - UP */

    CALL INITIALIZE$RCVR;
    CALL INITIALIZE$XMTR;
    CC$PTR1,CC$PTR2, USAR$CHECK=0;

    RECEIVE$LOGIC$PROCESS(DEVICE$READY)=FALSE;
    WRITER$PROCESS(DEVICE$READY)=FALSE;
    OPERATOR$INPUT$PROCESS(DEVICE$READY)=FALSE;
    OPERATOR$OUTPUT$PROCESS(SCHEDULED),
    OPERATOR$OUTPUT$PROCESS(DEVICE$READY)=FALSE;
    TRANSMITTER$PROCESS(DEVICE$READY)=FALSE;

    OUTPUT(TTY$STATUS)=1;  /* STROBE TTY READER */
    OUTPUT(TTY$STATUS)=0;

    /* INITIALIZE TEST/DEBUG BUFFERS AND POINTERS, ETC. */

    R$TEST$PTR, BYTE$COUNTER, TEST$COUNTER=0;

    /* INITIALIZE PAPER BUFFER */

    DO PAPER$PTR=0 TO 400;
        PAPER(PAPER$PTR)=0;
    END;
    PAPER$PTR=0;
```

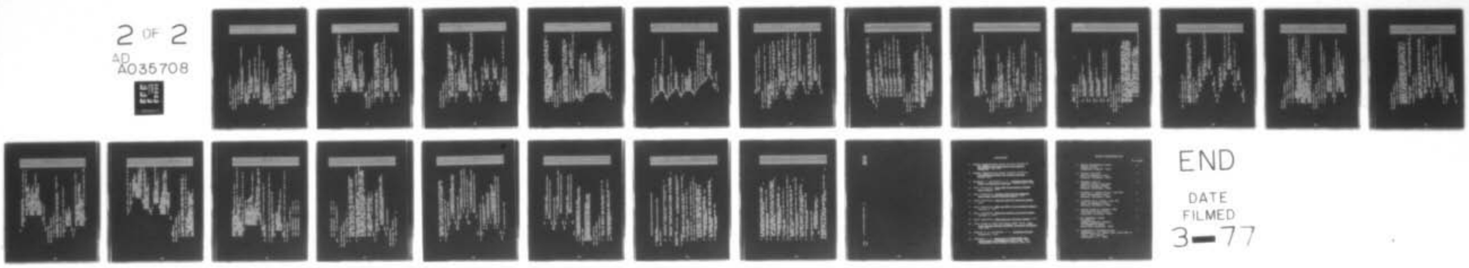
```
    S$TEST$PTR=7;
    T$TEST$PTR=0;
     SENDING=TRUE;

    /* END OF TEST DEBUG INITIALIZATION */
END INITIALIZE;

POLL$USAR: PROCEDURE;

    /*    LEVEL 2 PROCEDURE    */

    /* POLLS THE RECEIVE SIDE OF THE USART TO SEE IF A RECEIVED
          BYTE IS READY FOR PROCESSING.    */


    IF INPUT(USART$STATUS) THEN /* USAR NOT READY */
          RECEIVE$LOGIC$PROCESS(DEVICE$READY)=FALSE;
    ELSE /* DEVICE IS READY */
          RECEIVE$LOGIC$PROCESS(DEVICE$READY)=TRUE;



IF VIRTUAL THEN
    RECEIVE$LOGIC$PROCESS(DEVICE$READY)=TRUE;

END POLL$USAR;

RECEIVE$LOGIC: PROCEDURE;

    /*    LEVEL 2 PROCEDURE    */

    /*   EXAMINES INCOMING (RECEIVED) BYTES ONE AT A TIME.
          DETERMINES LOSS OR GAIN OF SYNCHRONOUS IDLE, CHECKS
          FOR ODD/EVEN PARITY, RECOGNIZES AND ACTS UPON CONTROL
          CHARACTERS, SCHEDULES THE TRANSMITTER PROCESS WHEN
          NECESSARY, PLACES TEST BYTES IN BUFFERS, SCHEDULES
          THE WRITER PROCESS WHEN THESE BUFFERS ARE READY FOR
          WRITING, SELECTS THE CORRECT WRITER (OUTPUT) DEVICE
          BY EXAMINING THE 'SEL' CHARACTER, SETS TIMERS AND
          LOGICAL FLAGS AND VARIABLES BASED UPON RECEIVED
          INFORMATION, SCHEDULES OPERATOR CONSOLE PROCESS
          WHENEVER ALARM CONDITIONS OCCUR, AND SCHEDULES ITSELF
          WHENEVER SYNCHRONIZATION MUST BE ACHIEVED.

         THE RECEIVE LOGIC PROCEDURE IS A 9-STATE AUTOMATON
         WHICH HAS THE FOLLOWING 9 STATES:

             1. WAITING FOR SOH.
```

```
        2. WAITING FOR STX.
        3. PROCESSING TEXT.
        4. WAITING FOR ETB.
        5. ATTEMPTING TO ACHIEVE SYNCHRONIZATION.
        6. EXPECTING SEL.
        7. EXPECTING DEL.
        8. EXPECTING BLOCK PARITY.
        9. EXPECTING SECOND CHARACTER OF 2-CHARACTER CONTROL
           SEQUENCE.*/

DECLARE
    LOOKUP BYTE,      /* USED TO INDEX ACTION LOOKUPS */
    ACTION BYTE;        /* THE ACTION 1-13 PERFORMED */

PERFORM$ACTION: PROCEDURE;

/* LEVEL 3 PROCEDURE */

/* PERFORMS ONE OF 13 ACTIONS FOR RECEIVER STATES 1-4 */


    DO CASE ACTION;

        ;/* CASE 0 NOT USED */

        DO;/* CASE 1: EXPECTED SOH, GOT SOH */
            RCV$STATE=6;
            RCV$ERROR, CHECK$RCV$TIMER=FALSE;
           BLOCK$START$STATE=1;
        END;

            /* CASE 2: DIDN'T GET EXPECTED CHARACTER */
        SYN$CTR=0;

        DO;/* CASE 3: FIRST OF 2-CHARACTER CONTROL SEQUENCE */
            SYN$CTR=0;
            PREV$RCV$STATE=RCV$STATE;/* SAVE PREVIOUS STATE */
            PREV$RCV$BYTE=RCV$BYTE;/* SAVE PREVIOUS BYTE */
            RCV$STATE=9;
        END;

        DO;/* CASE 4: SYN RECEIVED BETWEEN LINE BLOCKS */
            SYN$CTR=SYN$CTR + 1;
            IF SYN$CTR > 3 THEN
            SYN$CTR,RCV$SYNCH$TIMER=0;
        END;

        DO;/* CASE 5: EXPECTED STX, GOT STX */
```

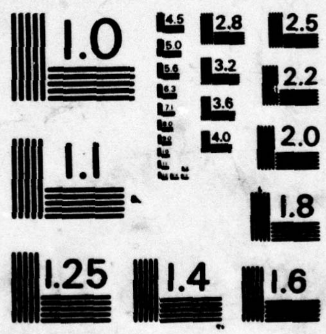
```
        CHECK$RCV$TIMER=FALSE;
        BLOCK$START$STATE=2;
        RCV$STATE=7;
    END;

    DO;/* CASE 6: PROCESSING TEXT, INVAIID CHAR RECEIVED */
        RCV$TXT$CTR=RCV$TXT$CTR + 1;
        RCV$ERROR=TRUE;
        IF RCV$TXT$CTR > 80 THEN
        RCV$STATE=4;
    END;

    DO;  /* CASE 7: PROCESSING TEXT, RECEIVE EM */
        RCV$BP=RCV$BP + RCV$BYTE;
        RCV$STATE=4;
    END;

    DO; /* CASE 8: PROCESSING TEXT, MC RECEIVED */
        MC$FLAG=TRUE;
        RCV$BP=RCV$BP + RCV$BYTE;
    END;

    DO;  /* CASE 9: PROCESSING TEXT, GET TEXT CHARACTER */
        RCV$BP=RCV$BP + RCV$BYTE;
        RCV$TXT$CTR=RCV$TXT$CTR + 1;
        RCV$BUFFER(RCV$TXT$CTR)=RCV$BYTE AND LOW$7$MASK;
        IF RCV$TXT$CTR=80 THEN RCV$STATE=4;
        /* MASK REMOVES LEFTMOST PARITY BIT */
    END;

    DO;  /* CASE 10: EXPECTING ETB/ETX, GET ERRONEOUS CHAR */
        RCV$ERROR=TRUE;
        RCV$STATE=8;
    END;

    DO;  /* CASE 11: EXPECTING ETB, GET ETX */
        RCV$MID$MSG,MC$FLAG=FALSE;
        RCV$BP=RCV$BP + RCV$BYTE;
        RCV$STATE=8;
    END;

    ; /* CASE 12: GET SYN AT END OF BLOCK - DO NOTHING */

    DO; /* CASE 13: EXPECTING ETB/ETX, GET ETB */
        RCV$BP=RCV$BP + RCV$BYTE;
        RCV$STATE=8;
    END;
```

AD-A035 708 NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF F/G 17/2
THE USE OF MICROCOMPUTERS IN DCS AUTODIN TRIBUTARIES.(U)
DEC 76 G E ANDERSON
UNCLASSIFIED NL

2 OF 2
AD
A035708



END
DATE
FILMED
3-77



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

```
    END;    /* OF CASE ACTION */

END PERFORM$ACTION;

RCV$STATE$5: PROCEDURE;

     /* LEVEL 3 PROCEDURE */

     /* WHEN THE RECEIVER IS IN STATE 5 IT IS ATTEMPTING
         TO ACHIEVE SYNCHRONIZATION */

      RCV$SYNCH$TIMER=0;
     SYN$CTR=0;
     DO WHILE SYN$CTR < 4;
         DO WHILE NOT (RECEIVE$LOGIC$PROCESS
              (DEVICE$READY));
             CALL POLL$USAR;
         END;
         IF (RCV$BYTE:=GET$RCVD$BYTE)=SYN THEN
             SYN$CTR=SYN$CTR + 1;
         ELSE SYN$CTR = 0;
          IF RCV$SYNCH$TIMER EXPIRED THEN CALL ALARM(10);
     END;

     RECEIVE$LOGIC$PROCESS(SCHEDULED)=FALSE;
     RCV$SYNCH$TIMER=0;
     CHECK$RCV$TIMER=TRUE;
     IF RCV$MID$MSG THEN RCV$STATE=2;
     ELSE RCV$STATE=1;
END RCV$STATE$5;

RCV$STATE$6: PROCEDURE;

     /* LEVEL 3 PROCEDURE */

     /* WHEN THE RECEIVER IS IN STATE 6 IT IS EXPECTING
     TO GET AN 'SEL' CHARACTER.  IF THE SEL CHARACTER IS
     CORRECT (BASED ON A TABLE LOOKUP), THE RECEIVER GOES
     TO STATE 3 (TO PROCESS TEXT); OTHERWISE, IT RETURNS
     TO STATE 1 (WAITING FOR SOH). */

     IF ((RCV$BYTE:=RCV$BYTE AND LOW$7$MASK) < 'A' OR
         RCV$BYTE > 'D') AND RCV$BYTE <> 'H' AND
         RCV$BYTE <> 'F' AND RCV$BYTE <> 'S' THEN
         /* WE HAVE A BAD 'SEL' CHARACTER */
         DO;
             CHECK$RCV$TIMER=TRUE;
             RCV$SYNCH$TIMER=0;
```

```
            RCV$STATE=1;
        END;
    ELSE /* GOOD 'SEL' CHARACTER RECEIVED */
        DO;
            RCV$BP=(RCV$BP:=0) + RCV$BYTE; /* INITIALIZE BP */
            IF RCV$BYTE='A' OR RCV$BYTE='H' OR RCV$BYTE='S' THEN
            /* WE WANT TO OUTPUT ON THE TELETYPE */
                OUTPUT$DEVICE=TTY;
            ELSE
                IF RCV$BYTE='D' OR RCV$BYTE='F' THEN
                /* THEN IT'S A CARD MESSAGE */
                    OUTPUT$DEVICE=CARD$PUNCH;
                ELSE /* IT MUST BE A MAG TAPE MESSAGE */
                    OUTPUT$DEVICE=MAG$TAPE;
            RCV$TXT$CTR=0; /* INITIALIZE FOR NEW LINE BLOCK */
             RCV$MID$MSG=TRUE;
            RCV$ERROR=FALSE;
            RCV$STATE=3;

            /* CHECK FOR FLASH MESSAGE - RING BELL
            FOR OPERATOR IF FLASH */
            IF RCV$BYTE='F' OR RCV$BYTE='S' THEN
                CALL ALARM(9);
        END;
END RCV$STATE$6;

RCV$STATE$7: PROCEDURE;

    /* LEVEL 3 PROCEDURE */

    /* WHEN THE RECEIVER IS IN STATE 7 IT IS EXPECTING A
        'DEL' CHARACTER.  IF DEL RECEIVED THEN GOES INTO
        STATE 3 (PROCESSING TEXT) OTHERWISE, IT GOES INTO
        STATE 2 (WAITING FOR STX CHARACTER).  */

    IF RCV$BYTE=DEL THEN /* GO TO TEXT PROCESSING STATE */
        DO;
            RCV$ERROR=FALSE;     /* INITIALIZE FOR NEW BLOCK */
            RCV$TXT$CTR, RCV$BP=0;
            RCV$BP=RCV$BP + RCV$BYTE;
            RCV$STATE=3;
        END;
    ELSE /* BYTE IS NOT A DEL - GO BACK TO STATE 2 */
        DO;
            RCV$SYNCH$TIMER=0;
            CHECK$RCV$TIMER=TRUE;
            RCV$STATE=2;
        END;
```

```
END RCV$STATE$7;

RCV$STATE$8: PROCEDURE;

    /* LEVEL 3 PROCEDURE */

    /* IN STATE 8 THE RECEIVER IS LOOKING FOR VALID BLOCK
        PARITY.  IF BP IS VALID, RECEIVER FINISHES THE
        BLOCK AND ACKS FOR IT; OTHERWISE IT SENDS A
        NAK (NEGATIVE ACKNOWLEDGEMENT           */

    IF RCV$BP <> RCV$BYTE THEN     /* BAD PARITY */
        DO;
            CALL SEND$CC(NAK);
            NAK$CTR=NAK$CTR + 1;
            IF NAK$CTR > 3 THEN CALL ALARM(1);
            RCV$STATE=BLOCK$START$STATE;
            RCV$SYNCH$TIMER=0;
            CHECK$RCV$TIMER=TRUE;
            RETURN;
        END;
     IF W$BUFFER$PTR > 1 THEN STILL$WRITING=TRUE;
    IF STILL$WRITING AND RCV$MID$MSG THEN /* WRITER PROCESS IS NOT
                                            FINISHED WITH LAST BLOCK */
        DO;
            CALL SEND$CC(WBT);
            NAK$CTR=0;
        END;
    ELSE
        DO;
            IF XMT$ACK$1$2=1 THEN
                DO;
                    CALL SEND$CC(ACK1);
                    NAK$CTR=0;
                    XMT$ACK$1$2=2;
                END;
            ELSE
                DO;
                    CALL SEND$CC(ACK2);
                    NAK$CTR=0;
                    XMT$ACK$1$2=1;
                END;
        END;
    RCV$SYNCH$TIMER=0;
    CHECK$RCV$TIMER=TRUE;
    IF RCV$MID$MSG THEN RCV$STATE=2;    /* SET NEXT RCV STATE */
    ELSE RCV$STATE=1;
    WRITER$PROCESS(SCHEDULED)=TRUE;
```

```
    /* THIS SCHEDULES THE WRITER PROCESS TO DUMP THE BUFFER */
    RCV$BUFFER=RCV$TXT$CTR;  /* INSERT BUFFER LENGTH IN SLOT 1 */
    /* NOW SWITCH BUFFERS FOR THE RECEIVER LOGIC PROCESS */
    IF RCV$BUFFER$ADDR=.RW$BUFFER$1 THEN
         RCV$BUFFER$ADDR=.RW$BUFFER$2;
    ELSE RCV$BUFFER$ADDR=.RW$BUFFER$1;

END RCV$STATE$8;

RCV$STATE$9: PROCEDURE;

    /* LEVEL 3 PROCEDURE */

    /* IN STATE 9 THE RECEIVER IS EXPECTING THE SECOND OF A
    TWO-CHARACTER CONTROL SEQUENCE. IF THIS IS RECEIVED, THE
    CONTROL CHARACTERS ARE ACTED UPON, OTHERWISE AN ERROR
    CONDITION IS FLAGGED (RCV$ERROR).            */

    RCV$STATE=PREV$RCV$STATE;        /* NORMAL RETURN IS TO
                                     PREVIOUS STATE */
    IF PREV$RCV$BYTE <> RCV$BYTE THEN /* NOT A VALID CONTROL SEQ */
      DO;
        RCV$ERROR=TRUE;
        RETURN;
      END;
    IF RCV$BYTE=ACK1 OR RCV$BYTE=ACK2 THEN
    /* WE HAVE A VALID CONTROL CHAR SEQUENCE */
      DO;
        IF (RCV$BYTE=ACK1 AND RCV$ACK$1$2=1) OR
          (RCV$BYTE=ACK2 AND RCV$ACK$1$2=2) THEN
          DO;
            AWAITING$ACK,XMT$WAIT=FALSE;
            WBT$CTR,NAK$CTR,XMT$REP$CTR=0;
            CHECK$ANS$TIMER=FALSE;
            XMT$ANS$TIMER=0;
            /* RESET TOGGLE */
            IF RCV$ACK$1$2=1 THEN RCV$ACK$1$2=2;
            ELSE RCV$ACK$1$2=1;
            T$BUFFER$PTR=1;  /* RESET POINTER FOR REXMT */
            TRANSMIT$BUFFER=0;  /* MARKS BUFFER AS ACK'D */
            /* NOW SWITCH BUFFERS FOR TRANSMITTER PROCESS */
            IF TRANSMIT$BUFFER$ADDR=.T$BUFFER$1 THEN
              TRANSMIT$BUFFER$ADDR=.T$BUFFER$2;
            ELSE TRANSMIT$BUFFER$ADDR=.T$BUFFER$1;
          END;
        ELSE
          CALL SEND$CC(INV); /* ACK1/2 SEQUENCE WRONG */
      END;
```

```
ELSE
  IF RCV$BYTE=NAK THEN
    DO;
      IF NAK$CTR > 3 THEN CALL ALARM(1);
      ELSE
        DO;
          XMT$ANS$TIMER=0;
          NAK$CTR=NAK$CTR + 1;
          T$BUFFER$PTR=1;  /* RESETS FOR RETRANSMISSION */
        END;
    END;
  ELSE
    IF RCV$BYTE=WBT THEN
      DO;
        IF WBT$CTR > 3 THEN CALL ALARM(2);
        ELSE
          DO;
            WBT$CTR=WBT$CTR + 1;
            XMT$ANS$TIMER=0;
          END;
      END;
    ELSE
      IF RCV$BYTE=CAN THEN
        DO;
          CALL ALARM(3);
          RCV$STATE=1;
          XMT$ACK$1$2=1;
          CHECK$RCV$TIMER=TRUE;
          RCV$SYNCH$TIMER=0;
        END;
      ELSE
        IF RCV$BYTE=REP THEN
          DO;
            IF RCV$MID$MSG OR STILL$WRITING THEN
              DO;
                IF STILL$WRITING THEN CALL SEND$CC(WBT);
                ELSE
                  IF NAK$CTR > 0 THEN CALL SEND$CC(NAK);
                  ELSE
                    DO;
                      IF XMT$ACK$1$2=1 THEN
                        CALL SEND$CC(ACK2);
                      ELSE CALL SEND$CC(ACK1);
                    END;
              END;
            ELSE
              CALL ALARM(4);
          END;
```

```
            ELSE
              IF RCV$BYTE=RM THEN CALL ALARM(5);
              ELSE
                CALL ALARM(6);  /* INV WAS RECEIVED */

END RCV$STATE$9;

/*   ***** RECEIVER LOGIC BEGINS EXECUTION HERE    ***** */

RCV$BYTE=GET$RCVD$BYTE;
IF RCV$STATE < 5 THEN
    DO; /* FIRST DETERMINE ACTION BASED ON RCVD BYTE AND
            RCV STATE */
        RCV$BYTE=RCV$BYTE AND 0FFH; /* PERFORM AND TO INSURE
                                   THAT PARITY IS SET */
        IF PARITY EVEN THEN
            DO;
                IF(LOOKUP:=(RCV$BYTE AND LOW$7$MASK))>7 THEN
                    IF(LOOKUP:=LOOKUP-9) > 21 THEN
                        LOOKUP=20;
                /* THIS HAS MAPPED THE CONTROL CHARACTERS INTO
                    THE INTEGERS 1-21 */
            END;
        ELSE LOOKUP=22;  /* BYTE IS NOT CONTROL CHARACTER */
        /* PERFORM TABLE LOOKUP BASED ON RCV STATE AND CHARACTER */
        DO CASE RCV$STATE;
            ; /* CASE 0 NOT USED */
            DO; /* CASE 1 - CHECK FOR LOSS OF SYNCH. IF SYNCH
                 OK THEN DO LOOKUP FOR RCV STATE 1 */
                IF CHECK$RCV$TIMER AND RCV$SYNCH$TIMER EXPIRED THEN
                    DO;
                        RCV$STATE=5;
                        RECEIVE$LOGIC$PROCESS(SCHEDULED)=TRUE;
                        RETURN;
                    END;
                ACTION=TABLE$1(LOOKUP);
            END;

            DO; /* CASE 2 - CHECK FOR SYNC LOSS.  IF SYNCH
                 OK THEN DO LOOKUP FOR RCV STATE 2 */
                IF CHECK$RCV$TIMER AND RCV$SYNCH$TIMER EXPIRED THEN
                    DO;
                        RCV$STATE=5;
                        RECEIVE$LOGIC$PROCESS(SCHEDULED)=TRUE;
                        RETURN;
                    END;
                ACTION=TABLE$2(LOOKUP);
            END;
```

```
            ACTION=TABLE$3(LOOKUP);  /* CASE 3 - RCV STATE 3 */
            ACTION=TABLE$4(LOOKUP); /* CASE 4 - RCV STATE 4  */
        END; /* OF CASE RCVSTATE */
        CALL PERFORM$ACTION;
    END;

ELSE       /* RECEIVER STATE IS BETWEEN 5 AND 9 */
    DO CASE (ACTION:=RCV$STATE - 5);

            /* CASE 0 - RCV$STATE =5 - ATTEMPTING TO
                    ACHIEVE SYNCHRONIZATION */
        CALL RCV$STATE$5;

        /* CASE 1, RCV$STATE = 6 - EXPECTING 'SEL' BYTE */
        CALL RCV$STATE$6;

        /* CASE 2, RCV$STATE=7 - EXPECTING 'DEL' BYTE */
        CALL RCV$STATE$7;

        /* CASE 3, RCV$STATE=8 - EXPECTING BLOCK PARITY BYTE */
        CALL RCV$STATE$8;

        /* CASE 4, RCV$STATE=9 - PROCESSING THE SECOND CHAR
        IN A TWO-CHARACTER CONTROL SEQUENCE */
        CALL RCV$STATE$9;

    END; /* OF CASE ACTION */

END RECEIVE$LOGIC;

WRITER: PROCEDURE;

    /*   LEVEL 2 PROCEDURE   */

    /*   WRITES BYTES (ONE AT A TIME) ON THE SELECTED
         OUTPUT DEVICE AND RESCHEDULES ITSELF WHENEVER
         MORE BYTES ARE TO BE DUMPED FROM A WRITE BUFFER. */

    CALL WRITE$BYTE(WRITE$BUFFER(W$BUFFER$PTR));
    W$BUFFER$PTR=W$BUFFER$PTR + 1;
    IF W$BUFFER$PTR > WRITE$BUFFER THEN /* DONE WRITING CURRENT BUFF */
        DO;
            WRITE$BUFFER=0;
            /* SET LENGTH OF BUFFER- IT'S ALREADY WRITTEN.
                THEN FLIP BUFFERS      */
            IF WRITE$BUFFER$ADDR=.RW$BUFFER$1 THEN
                WRITE$BUFFER$ADDR=.RW$BUFFER$2;
```

```
            ELSE WRITE$BUFFER$ADDR=.RW$BUFFER$1;
             W$BUFFER$PTR=1;
             STILL$WRITING=FALSE;
            IF WRITE$BUFFER=0 THEN /* GO TO SLEEP - NO
                                      RESCHEDULE NEEDED */
              DO;
                 IF NOT RCV$MID$MSG THEN OUTPUT$DEVICE=NOT$SELECTED;
                 WRITER$PROCESS(SCHEDULED)=FALSE;
              END;
        END;
END WRITER;

OPERATOR$INPUT: PROCEDURE;

    /*   LEVEL 2 PROCEDURE   */

    /* INTERPRETS COMMANDS FROM THE OPERATOR CONSOLE -
        T= TRANSMIT NEW MESSAGE.
        C= CANCEL CURRENTLY TRANSMITTED MSG.
        B= REBOOT (RE-INITIALIZE).     */

    DECLARE CHARACTER BYTE;

    RETURN;  /* INSERTED FOR TEST/DEBUG ONLY */
    IF(CHARACTER:=CRT$IN)='T' THEN  /* TRANSMIT NEW MSG */
        SENDING=TRUE;
    ELSE
        IF CHARACTER='C' THEN  /* CANCEL THIS MSG */
            DO;
                CALL SEND$CC(CAN);
                CALL INITIALIZE$XMTR;
            END;
        ELSE
            IF CHARACTER='B' THEN  /* REBOOT THE SYSTEM */
                GO TO RESTART;
            /* ELSE NOTHING - ONLY T,C, AND B VALID COMMANDS */
END OPERATOR$INPUT;

OPERATOR$OUTPUT: PROCEDURE;

    /*   LEVEL 2 PROCEDURE   */

    /* OUTPUTS MESSAGES TO THE CRT ONE BYTE AT A TIME:

        STATE 1: SEND BELL
        STATE 2: SEND ONE LETTER DIAGNOSTIC MSG.
```

```
        STATE 3: SEND CR.
        STATE 4: SEND LF                                    */

    DO CASE OUTPUT$STATE;

        ;    /* CASE 0 NOT USED */

        DO;  /* CASE 1 OUTPUT BELL ALARM */
             CALL CRT$OUT(BELL);
             OUTPUT$STATE=2;
        END;

        DO;  /* CASE 2 - OUTPUT DIAGNOSTIC CHARACTER */
             CALL CRT$OUT(ALARM$MSG);
             OUTPUT$STATE=3;
        END;

        DO;  /* CASE 3 - OUTPUT <CR> */
             CALL CRT$OUT(CR);
             OUTPUT$STATE=4;
        END;

        DO;  /* CASE 4 - OUTPUT <LF> AND GO TC SLEEP */
             CALL CRT$OUT(LF);
             OPERATOR$OUTPUT$PROCESS(SCHEDULED)=FALSE;
        END;

    END; /* OF CASE OUTPUT STATE */

END OPERATOR$OUTPUT;

TRANSMIT$LOGIC: PROCEDURE;

    /*   LEVEL 2 PROCEDURE   */

    /*  BUILDS BLOCKS FOR OUTGOING MESSAGE TRANSMISSION (ONE
        BYTE AT A TIME).  INSERTS NEEDED FRAMING BYTES (SOH,STX,
        SEL, DEL, EM, ETB, ETX, AND BP), CALCULATES ODD PARITY
        FOR TEXT BYTES, SCHEDULES TRANSMITTER PROCESS WHEN LINE
        BLOCKS BECOME AVAILABLE FOR TRANSMISSION, AND SCHEDULES
        THE OPERATOR OUTPUT PROCESS WHEN ALARM CONDITIONS OCCUR.
        THE TRANSMIT LOGIC PROCESS IS A 5 STATE AUTOMATON WITH
        THE FOLLOWING STATES:
             1. PROCESSING BEGINNING OF A NEW MESSAGE.
             2. WAITING FOR SECOND CHARACTER OF A MESSAGE.
             3. WAITING FOR THIRD CHARACTER OF A MESSAGE.
             4. PROCESSING TEXT IN THE MIDDLE OF A MESSAGE.
             5. PROCESSING THE BEGINNING OF AN INTERMEDIATE LINE BLOCK.
```

```
     */
EOM: PROCEDURE(CHARACTER) BYTE;

     /* LEVEL 3 PROCEDURE */

     /* CHECKS FOR 8 LINE FEEDS FOLLOWED BY 4 N'S IN A ROW.
     RETURNS TRUE ONLY IF THIS HAS OCCURED (THIS IS THE END OF A
     MESSAGE IN AUTODIN).  OTHERWISE, IT RETURNS FALSE.    */

     DECLARE CHARACTER BYTE;

     IF EOM$STATE=1 THEN
          DO;
               IF CHARACTER=08AH THEN
                /* 08AH=ODD PARITY  LF */
                    DO;
                         LF$CTR=LF$CTR + 1;
                         IF LF$CTR=8 THEN /* WE HAVE 8 LF'S IN A ROW
                              GO TO EOM$STATE 2 AND COUNT N'S */
                              DO;
                                   LF$CTR=0;
                                   EOM$STATE=2;
                              END;
                    END;
               ELSE LF$CTR=0;
          END;
     ELSE
          DO;
               IF CHARACTER=0CEH THEN
                /* 0CEH=ODD PARITY 'N' */
                    DO;
                         N$CTR=N$CTR + 1;
                         IF N$CTR =4 THEN /* WE'RE AT END OF MSG */
                              DO;
                                   N$CTR=0;
                                   EOM$STATE=1;/* RESET NEXT TIME */
                                   RETURN TRUE;
                              END;
                    END;
               ELSE  /* NOT AN EOM - GO BACK TO STATE 1  */
                    DO;
                         N$CTR=0;
                         EOM$STATE=1;
                    END;
          END;
     RETURN FALSE;
```

```
END EOM;

XMT$STATE$1: PROCEDURE;

    /* LEVEL 3 PROCEDURE */

    /*  STATE 1 IS WAITING TO START A NEW MSG FOR TRANSMISSION  */

    IF (XMT$BYTE:=GET$BYTE$TO$XMT)=SPACE THEN /* GOBBLE SPACES
                                            AT THE BEGINNING OF TAPE */
        RETURN;
    /* OTHERWISE GET A NEW BUFFER AND PUT AN 'SOH' IN SLOT 1 */
    IF XMT$LOGIC$BUFFER$ADDR=.T$BUFFER$1 THEN
        XMT$LOGIC$BUFFER$ADDR=.T$BUFFER$2;
    ELSE XMT$LOGIC$BUFFER$ADDR=.T$BUFFER$1;
    XMT$LOGIC$BUFFER(1)=SOH;
    XMT$BYTE=XMT$BYTE AND OFFH; /* ENSURE PARITY SET */
    IF PARITY EVEN THEN
        XMT$BYTE=XMT$BYTE OR PARITY$MASK; /* INSERT ODD PARITY */
    XMT$LOGIC$BUFFER(3)=XMT$BYTE;
    XMT$BP=(XMT$BP:=0) + XMT$BYTE; /* INITIALIZE AND COMPUTE BP */
    XMT$STATE=2; /* SET NEXT STATE */

END XMT$STATE$1;

XMT$STATE$2: PROCEDURE;

    /* LEVEL 3 PROCEDURE */

    /* PROCESSES THE SECOND BYTE OF A TRANSMITTED MESSAGE */

    IF CAN$FLAG THEN /* CANCEL OUTGOING MESSAGE */
        DO;
            CAN$FLAG=FALSE;
            XMT$STATE=1;
            CALL SEND$CC(CAN);
            SENDING=FALSE;
        END;
    ELSE /* PROCESS SECOND BYTE OF MESSAGE */
        DO;
            XMT$BYTE=((XMT$BYTE:=GET$BYTE$TO$XMT) AND OFFH);
            IF PARITY EVEN THEN
                XMT$BYTE=XMT$BYTE OR PARITY$MASK;
            XMT$LOGIC$BUFFER(4)=XMT$BYTE;
            XMT$BP=XMT$BP + XMT$BYTE;
            XMT$STATE=3; /* SET NEXT STATE */
        END;
```

```
END XMT$STATE$2;

XMT$STATE$3: PROCEDURE;

     /* LEVEL 3 PROCEDURE */

     /* CHECKS THE THIRD CHARACTER OF THE OUTGOING MESSAGE TO
     SEE IF IT IS A VALID LMF CHARACTER.  IF IT IS, IT IS
     CONVERTED TO THE CORRESPONDING 'SEL' CHARACTER.  THE LMF
     CHARACTER IS INSERTED IN THE THIRD TEXT SLOT OF THE
     LINE BLOCK; THE SEL CHARACTER IS INSERTED IN THE SECOND
     FRAMING SLOT OF THE LINE BLOCK.     */

     DECLARE SEL BYTE,         /* USED TO LOOK UP THE SEL CHAR */
          CHECK BYTE;          /* USED TO CHECK CORRECTNESS OF SEL */

     CHECK$LMF: PROCEDURE BYTE;

          /* LEVEL 4 PROCEDURE */

          /* RETURNS TRUE AND SETS THE SEL CORRECTLY IF THE LMF
          CHARACTER (THIRD BYTE OF THE OUTGOING MSG) WAS CORRECT.
          RETURNS FALSE OTHERWISE.     */

          IF XMT$BYTE > 40H AND XMT$BYTE < 4AH AND XMT$BYTE <> 'E'
               AND XMT$BYTE <> 'H' THEN
               DO;
                    SEL=SEL$LOOKUP$1(XMT$BYTE - 41H);
                    RETURN TRUE;
               END;
          ELSE
               IF XMT$BYTE > 50H AND XMT$BYTE < 55H THEN
                    DO;
                         SEL=SEL$LOOKUP$2(XMT$BYTE - 51H);
                         RETURN TRUE;
                    END;
               ELSE /* BAD LMF CHARACTER - NO CORRESPONDING SEL */
                    RETURN FALSE;
     END CHECK$LMF;

     IF CAN$FLAG THEN /* CANCEL MESSAGE */
          DO;
               CAN$FLAG=FALSE;
               XMT$STATE=1;
               CALL SEND$CC(CAN);
               SENDING=FALSE;
          END;
     ELSE
```

```
        DO;
            XMT$BYTE=GET$BYTE$TO$XMT;
            IF (CHECK:=CHECK$LMF) THEN /* GOOD LMF CHARACTER */
                DO;
                    SEL=SEL AND OFFH; /* ENSURE PARITY SET */
                    IF NOT (PARITY EVEN) THEN
                        SEL=SEL OR PARITY$MASK;
                    XMT$BP=XMT$BP + SEL;
                    XMT$LOGIC$BUFFER(2)=SEL;
                    XMT$BYTE=XMT$BYTE AND OFFH;
                    IF PARITY EVEN THEN
                        XMT$BYTE=XMT$BYTE OR PARITY$MASK;
                    XMT$BP=XMT$BP + XMT$BYTE;
                    XMT$LOGIC$BUFFER(5)=XMT$BYTE;
                    XMT$TXT$CTR=6;
                    XMT$STATE=4;
                END;
             ELSE CALL ALARM(8);
         END;

END XMT$STATE$3;

XMT$STATE$4: PROCEDURE;

    /* LEVEL 3 PROCEDURE */

    /* PROCESSES TEXT OF AN OUTGOING MESSAGE.  BUILDING THE
    INFORMATIONAL PORTION OF THE LINE BLOCKS.      */

    DECLARE CHECK BYTE;

    IF CAN$FLAG THEN /* CANCEL OUTGOING MESSAGE */
        DO;
            CAN$FLAG=FALSE;
            XMT$STATE=1;
            CALL SEND$CC(CAN);
            SENDING=FALSE;
        END;
    ELSE
        DO;
            XMT$BYTE=((XMT$BYTE:=GET$BYTE$TO$XMT) AND OFFH);
            IF PARITY EVEN THEN
                XMT$BYTE=XMT$BYTE OR PARITY$MASK;
            XMT$LOGIC$BUFFER(XMT$TXT$CTR)=XMT$BYTE;
            XMT$BP=XMT$BP + XMT$BYTE;
            XMT$TXT$CTR=XMT$TXT$CTR + 1;
            IF (XMT$TXT$CTR > 82) OR (CHECK:=EOM(XMT$BYTE)) THEN
                DO;
```

```
IF CHECK THEN /* WE'RE AT THE END OF MSG */
   DO;
      IF XMT$TXT$CTR < 82 THEN
           DO;
                XMT$LOGIC$BUFFER(
                     XMT$TXT$CTR)=EM;
                XMT$BP=XMT$BP + EM;
               SENDING=FALSE;
               XMT$TXT$CTR=XMT$TXT$CTR + 1;
               INPUT$DEVICE=NOT$SELECTED;
              IF VIRTUAL THEN SENDING=TRUE;
           END;
      XMT$STATE=1;/* SET STATE FOR NEW MSG*/
      XMT$LOGIC$BUFFER(XMT$TXT$CTR)=ETX;
      XMT$BP=XMT$BP + ETX;
      XMT$TXT$CTR=XMT$TXT$CTR + 1;
   END;
ELSE
      DO;
           XMT$STATE=5;
         XMT$LOGIC$BUFFER(XMT$TXT$CTR)=ETB;
           XMT$BP=XMT$BP + ETB;
           XMT$TXT$CTR=XMT$TXT$CTR + 1;
      END;
XMT$ANS$TIMER=0;
CHECK$ANS$TIMER=TRUE;
XMT$LOGIC$BUFFER(XMT$TXT$CTR)=XMT$BP;
XMT$LOGIC$BUFFER=XMT$TXT$CTR; /* INSERTS
        LENGTH AT BEGINNING OF BUFFER */
IF AWAITING$ACK THEN
        XMT$WAIT=TRUE;
                    END;
     END;
END XMT$STATE$4;

XMT$STATE$5: PROCEDURE;

    /* LEVEL 3 PROCEDURE */

    /* PROCESSING THE FRAMING CHARACTERS REQUIRED AT THE
    BEGINNING OF INTERMEDIATE LINE BLOCKS OF OUTGOING MSG */

    DECLARE CHECK BYTE ; /* USED TO CHECK FOR END OF MSG */
    /* GET NEW BUFFER */
    IF XMT$LOGIC$BUFFER$ADDR=.T$BUFFER$1 THEN
         XMT$LOGIC$BUFFER$ADDR=.T$BUFFER$2;
    ELSE XMT$LOGIC$BUFFER$ADDR=.T$BUFFER$1;
    XMT$LOGIC$BUFFER(1)=STX; /* INSERT 'STX' */
```

```
XMT$LOGIC$BUFFER(2)=DEL;
XMT$BYTE=((XMT$BYTE:=GET$BYTE$TO$XMT) AND OFFH);
IF PARITY EVEN THEN
     XMT$BYTE=XMT$BYTE OR PARITY$MASK; /* INSERT ODD PARITY */
XMT$LOGIC$BUFFER(3)=XMT$BYTE;
XMT$BP=(XMT$BP:=0) + DEL + XMT$BYTE;
 XMT$TXT$CTR=4;
IF (CHECK:=EOM(XMT$BYTE)) THEN /* WE'RE AT END OF A MSG */
     DO;
          XMT$LOGIC$BUFFER(4)=EM;
           SENDING=FALSE;
          XMT$LOGIC$BUFFER(5)=ETX;
          XMT$BP=XMT$BP + EM + ETX;
          XMT$LOGIC$BUFFER(6)=XMT$BP;
          XMT$LOGIC$BUFFER=6; /* STORE LENGTH OF BUFFER */
          XMT$ANS$TIMER=0;
          CHECK$ANS$TIMER=TRUE;
          IF AWAITING$ACK THEN XMT$WAIT=TRUE;
               /* TRANSMIT LOGIC PROCESS HAS 2 BLOCKS PENDING
               IT MUST WAIT UNTIL 1 OF THEM IS ACK'D.  */
          XMT$STATE=1;
     END;
ELSE XMT$STATE=4; /* CONTINUE PROCESSING TEXT */
END XMT$STATE$5;


/* ***** TRANSMIT LOGIC PROCESS BEGINS EXECUTION HERE ***** */

IF AWAITING$ACK AND XMT$ANS$TIMER EXPIRED AND
     CHECK$ANS$TIMER THEN
     DO;
          CALL SEND$CC(REP);
          XMT$ANS$TIMER=0;
          XMT$REP$CTR=XMT$REP$CTR + 1;
          IF XMT$REP$CTR > 3 THEN CALL ALARM(7);
          RETURN;
     END;
IF XMT$WAIT THEN RETURN; /* MUST SLEEP UNTIL A BLOCK IS ACK'D */

DO CASE XMT$STATE;

     ;    /* CASE 0 NOT USED */

     CALL XMT$STATE$1;   /* CASE 1*/

     CALL XMT$STATE$2;   /* CASE 2 */

     CALL XMT$STATE$3;   /* CASE 3 */
```

```
        CALL XMT$STATE$4;     /* CASE 4 */

        CALL XMT$STATE$5;     /* CASE 5 */

    END;       /* OF CASE XMT$STATE */

END TRANSMIT$LOGIC;

TRANSMITTER: PROCEDURE;

    /*   LEVEL 2 PROCEDURE   */

    /*  TRANSMITS BYTES ONE AT A TIME ACCORDING TO THE PRIORITY:
        1) CONTROL CHARACTER TO SEND, 2) LINE BLOCK DATA TO
        SEND, OR 3) SYN TO SEND.  IS AUTOMATICALLY SCHEDULED
        EVERY TIME THE USAT IS READY.       */

    IF (CC$PTR1 <> CC$PTR2) AND (T$BUFFER$PTR <> 2 OR T$BUFFER$PTR <>
        (T$BUFFER$LENGTH:=TRANSMIT$BUFFER)) THEN
        /* THE ABOVE LINES CHECK TO SEE IF THERE ARE CONTROL CHARACTERS
        TO SEND, CHECKS TO SEE THAT WE'RE NOT FRAMING A BLOCK, AND
        ASSIGNS BUFFER LENGTH THE VALUE STORED IN THE FIRST ELEMENT
        CF THE TRANSMIT BUFFER (WHERE TRANSMIT LOGIC PROCESS STORED
        THE LENGTH OF THE BUFFER TO TRANSMIT.    */

        DO; /* SEND A CONTROL CHARACTER */
        CALL SEND$BYTE(CONCHAR$BUFFER(CC$PTR1));
        IF (CC$PTR1:=CC$PTR1 + 1) > 7 THEN
            CC$PTR1=0;     /* RESET THE DEQUE */
        END;
    ELSE
        IF T$BUFFER$PTR <= TRANSMIT$BUFFER THEN
            /* SEND ONE BYTE OF THE LINE BLOCK */
            DO;
                IF T$BUFFER$PTR=1 THEN AWAITING$ACK = TRUE;
                CALL SEND$BYTE(TRANSMIT$BUFFER(T$BUFFER$PTR));
                T$BUFFER$PTR=T$BUFFER$PTR + 1;
            END;
        ELSE
            CALL SEND$BYTE(SYN);

END TRANSMITTER;

POLL$DEVICES: PROCEDURE;

    /*   LEVEL 2 PROCEDURE   */
```

```
/*  PERFORMED ONCE EVERY LOOP OF THE EXECUTIVE PROCESS
     SCHEDULER.  CHECKS TO SEE IF LOCAL I/O DEVICES ARE
     READY AND MARKS THEM READY FOR THE SCHEDULER.   */

IF OUTPUT$DEVICE <> NOT$SELECTED THEN
     DO;
          IF OUTPUT$DEVICE=TTY THEN
               DO;
                    IF NOT ROR(INPUT(1),2) THEN
                         WRITER$PROCESS(DEVICE$READY)=TRUE;
                    ELSE WRITER$PROCESS(DEVICE$READY)=FALSE;
               END;
          ELSE
          IF OUTPUT$DEVICE=MAG$TAPE THEN
                    DO;
                    IF NOT ROR(INPUT(3),2) THEN
                              WRITER$PROCESS(DEVICE$READY)=TRUE;
                         ELSE
                              WRITER$PROCESS(DEVICE$READY)=FALSE;
                    END;
               ELSE  /* CHECK THE CARD PUNCH FOR STATUS */
               DO;
                         IF NOT ROR(INPUT(4),2) THEN
                              WRITER$PROCESS(DEVICE$READY)=TRUE;
                         ELSE
                         WRITER$PROCESS(DEVICE$READY)=FALSE;
                    END;
     END;

IF INPUT$DEVICE=NOT$SELECTED THEN
     DO;
          IF NOT(INPUT(TTY$STATUS)) THEN
               INPUT$DEVICE=TTY;
          ELSE
               IF NOT(INPUT(MAG$TAPE)) THEN
                    INPUT$DEVICE=MAG$TAPE;
               ELSE
                    IF NOT(INPUT(CARD$PUNCH)) THEN
                         INPUT$DEVICE=CARD$PUNCH;
          IF INPUT$DEVICE=NOT$SELECTED THEN
               TRANSMIT$LOGIC$PROCESS(DEVICE$READY)=FALSE;
          ELSE
               TRANSMIT$LOGIC$PROCESS(DEVICE$READY)=TRUE;
     END;
ELSE
     DO;
          IF INPUT$DEVICE=TTY THEN
```

```
            DO;
                  IF NOT INPUT(1) THEN
                        WRITER$PROCESS(DEVICE$READY)=TRUE;
                  ELSE
                  WRITER$PROCESS(DEVICE$READY)=FALSE;
            END;
      ELSE
      IF INPUT$DEVICE=MAG$TAPE THEN
            DO;
                  IF NOT INPUT(3) THEN
                              WRITER$PROCESS(DEVICE$READY)=TRUE;
                        ELSE
                        WRITER$PROCESS(DEVICE$READY)=FALSE;
            END;
      ELSE
            DO;
                  IF NOT INPUT(4) THEN
                        WRITER$PROCESS(DEVICE$READY)=TRUE;
                        ELSE
                        WRITER$PROCESS(DEVICE$READY)=FALSE;
            END;
      END;

   IF NOT ROR(INPUT(CRT$STATUS),2) THEN
         OPERATOR$OUTPUT$PROCESS(DEVICE$READY)=TRUE;
   ELSE OPERATOR$OUTPUT$PROCESS(DEVICE$READY)=FALSE;

IF VIRTUAL THEN
   DO;
   WRITER$PROCESS(DEVICE$READY)=TRUE;
   TRANSMIT$LOGIC$PROCESS(DEVICE$READY)=TRUE;
   OPERATOR$INPUT$PROCESS(DEVICE$READY)=TRUE;
   OPERATOR$OUTPUT$PROCESS(DEVICE$READY)=TRUE;
   END;

END ECLL$DEVICES;

POLL$USAT: PROCEDURE;

   /* LEVEL 2 PROCEDURE */

   /*  POLLS THE TRANSMIT SIDE OF THE USART TO SEE IF THE
        USART IS READY TO SEND THE NEXT BYTE   */


   IF ROR(INPUT(USART$STATUS),2) THEN /* USAT NOT READY */
         TRANSMITTER$PROCESS(DEVICE$READY)=FALSE;
   ELSE  /* USAT IS READY */
```

```
        TRANSMITTER$PROCESS(DEVICE$READY)=TRUE;

IF VIRTUAL THEN
    TRANSMITTER$PROCESS(DEVICE$READY)=TRUE;

END PCLI$USAT;

    /*    **********************************************    */
    /*    ***********    PROGRAM LEVEL 1    ***********    */

    /*  EXECUTIVE SCHEDULER AND DEVICE MANAGER (NON-INTERRUPT
         VERSION).  FIRST EXECUTABLE STATEMENT IN THE PROGRAM.  */

VIRTUAL=TRUE;
RESTART:  CALL INITIALIZE;  /* INITIALIZE ALL PROGRAM VARIABLES
                               AND RETURN HERE FOR RESTART */

DO FOREVER;
    /* THE PROGRAM WILL LOOP AS LONG AS THE CPU IS FUNCTIONING
       AND ATTEMPT TO PROCESS AUTODIN MESSAGES                  */

    IF NOT VIRTUAL THEN /* SLOW DOWN FOR THE TTY */
        CALL TIME(250);

    CALL POLL$DEVICES; /* CHECK THE PERIPHERAL DEVICES */

    CALL POLL$USAR; /* CHECK THE RECEIVE SIDE OF THE USART */

    /* IF SYNCH HAS NOT BEEN ACHIEVED, THEN THE RECEIVE
       LOGIC PROCESS IS SCHEDULED TO ACHIEVE SYNCH.
       IF IT'S DEVICE IS READY THEN IT MUST PROCESS AN
       INCOMING BYTE NO MATTER WHAT IT'S STATUS IS  */

    IF RECEIVE$LOGIC$PROCESS (SCHEDULED) OR
        RECEIVE$LOGIC$PROCESS (DEVICE$READY) THEN
            CALL RECEIVE$LOGIC;

    /* IF INCOMING MESSAGE TRAFFIC NEEDS TO BE WRITTEN ON
       THE SELECTED OUTPUT DEVICE, THEN THE WRITER PROCESS
       WILL BE SCHEDULED                                    */

    IF WRITER$PROCESS(SCHEDULED) THEN
            CALL WRITER;
```

```
/* IF THE OPERATOR INPUT PROCESS DEVICE IS READY
   IT MEANS THE HUMAN OPERATOR DESIRES TO INPUT
   SOME INFORMATION TO THE PROGRAM                    */

IF CPERATOR$INPUT$PROCESS (DEVICE$READY) THEN
    CALL OPERATOR$INPUT;

/* IF AN ALARM NEEDS TO BE SENT TO THE HUMAN OPERATOR
    THEN THE OPERATOR OUTPUT PROCESS WILL BE SCHEDUL-
    ED.  IF THE DEVICE IS ALSO READY, THEN THE PROCESS
    IS PERMITTED TO RUN.                                  */

IF OPERATCR$OUTPUT$PROCESS (SCHEDULED) AND
    OPERATOR$OUTPUT$PROCESS (DEVICE$READY) THEN
         CALL OPERATOR$OUTPUT;

IF NOT VIRTUAL THEN /* SLOW DOWN FOR THE TTY */
    CALL TIME(250);

CALI POLL$USAT; /* CHECK THE TRANSMIT SIDE OF THE USART */

/* IF TRANSMIT LOGIC'S DEVICE IS READY AND WE ARE
   SENDING A MESSAGE, THEN WE MUST PROCESS AN
   CUTGOING BYTE.                                      */

IF TRANSMIT$LOGIC$PROCESS(DEVICE$READY) AND SENDING THEN
    CALL TRANSMIT$LOGIC;

/* IF TRANSMIT PROCESS'S DEVICE IS READY THEN WE
   MUST SEND SOMETHING TO THE TRANSMIT SIDE OF THE
   USART, EVEN IF IT IS ONLY SYN PATTERN               */

IF TRANSMITTER$PROCESS (DEVICE$READY) THEN
         CALL TRANSMITTER;


IF EYTE$COUNTER=60000 THEN
    DO;
         BYTE$COUNTER=0;
         TEST$COUNTER=TEST$COUNTER + 1;
         IF TEST$COUNTER=3 THEN HALT;
    END;

 USAR$CHECK=USAR$CHECK + 1;
 IF USAR$CHECK > 250 THEN CALL ALARM(11);
 /* WE'VE LOOPED 250 TIMES WITHOUT GETTING A RECEIVED BYTE -
 SOMETHING IS OBVIOUSLY WRONG - - CALL ALARM.                 */
```

```
END;      /* OF THE DO FOREVER LOOP  */                    AUT17770
                                                          AUT17780
EOF                                                       AUT17790
```

BIBLIOGRAPHY

1. Defense Communications Agency Circular 310-D70-30, DCS AUTODIN Switching Center and Tributary Operations, June 1970.

2. Defense Communications Agency Circular 370-D175-1, DCS AUTODIN Interface and Control Criteria, October 1970.

3. Hartmanis, J. and Stearns, R. E., Algebraic Structure Theory of Sequential Machines, Prentice-Hall, 1966.

4. Intel Corporation, Intel 8080 Microcomputer Systems User's Manual, 1976.

5. Intel Corporation, Intellec 8/Mod 80 Microcomputer Development System Reference Manual, 1974.

6. Intel Corporation, Intellec 8/Mod 80 Operators Manual, 1974.

7. Intel Corporation, 8008 and 8080 PL/M Programming Manual, revision A, 1975.

8. Intel Corporation, 8080 PL/M Compiler Operators Manual, revision A, 1975.

9. Intel Corporation, 8080 Simulator Software Package, 1974.

10. Joint Chiefs of Staff Publication JANAP 128(E), Automatic Digital Network (AUTODIN) Operating Procedures, June 1973.

11. Madnick, S. E. and Donovan, J. J., Operating Systems, McGraw-Hill, 1974.

12. Renninger, J. F., Reduction of AUTODIN/NIDN Line Disciplines to Programmed Control Logic, Naval Postgraduate School Master's Thesis, June 1975.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93940	1
4.	Professor V. Michael Powers, Code 52Pw Naval Postgraduate School Monterey, California 93940	2
5.	Professor Uno R. Kodres, Code 52Kr Naval Postgraduate School Monterey, California 93940	1
6.	Captain Gordon E. Anderson, USMC 2125 Mountain Vista Drive Encinitas, California 92024	1
7.	Mr. Kenneth R. Ausich Building 110 Stanford Research Institute 333 Ravenswood Avenue Menlo Park, California 94025	1
8.	Commandant of the Marine Corps Headquarters, U.S. Marine Corps (Code CCPP-11) (Attn: Major Beeler) Washington, D. C. 20380	1